VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Stabilizace obrazu

# Digital Image Stabilization

2010 Štěpán Sojka

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Štěpán Sojka**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Stabilizace obrazu
Digital Image Stabilization

Zásady pro vypracování:

Cílem práce je implementovat algoritmus (případně několik algoritmů) pro stabilizaci obrazu snímaného kamerou, která se nežádoucím způsobem chvěje vlivem větru nebo okolo jedoucích vozidel. Zásadním požadavkem práce je, aby byla výsledná implementace schopna tuto stabilizaci provádět v reálném čase. Předpokládá se použití vícevláknového zpracování a případné využití SIMD instrukcí.

1. Seznamte se s dostupnými algoritmy pro stabilizaci obrazu.
2. Vybraný stabilizační algoritmus implementujte.
3. Na poskytnuté testovací sekvenci ověřte pomocí vhodné metriky kvalitu implementovaného stabilizačního algoritmu.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Tomáš Fabián**

Datum zadání: 20.11.2009

Datum odevzdání: 07.05.2010

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA

doc. Dr.Ing. Eduard Sojka
*vedoucí katedry*

prof. Ing. Ivo Vondrák, CSc.
*děkan fakulty*

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2010 .....

Rád bych na tomto místě poděkoval vedoucímu práce, panu Ing. Tomášovi Fabiánovi, za cenné rady a pomoc, které mi poskytl při její tvorbě.

## Abstrakt

Tato práce se zabývá digitální stabilizací obrazu. Cílem stabilizace obrazu je odstranit z videosekvence chvění způsobené nechtěným pohybem kamery. To je užitečné jednak pro zlepšení subjektivní kvality záznamu, ale také pokud se videosekvence dále automaticky zpracovává. V této práci je prezentován systém pro digitální stabilizaci obrazu, který pracuje reálném čase a s nízkými nároky na výpočetní výkon. Díky tomu je možné použít prezentovaný systém jako předstupeň pro další zpracování videosekvence, například počítání pohybujících se objektů. Jak úspěšnost tak i rychlost stabilizace jsou ověřeny na testovacích videosekvencích.

**Klíčová slova:** stabilizace obrazu, paralelní programování, Streaming SIMD Extensions

## Abstract

This thesis is focused on digital image stabilization. The goal of image stabilization is to remove the unwanted jitter from a video sequence while preserving the intentional movements of the camera. This is not only useful for enhancing the visual quality of the video but also for improving the performance of other systems that may further analyze the sequence. The system presented in this thesis works in real time and with low processing power requirements: therefore it can be used as a front end preprocessor for other video analysis algorithms, for example, the algorithms for counting moving objects. Both accuracy and speed of the system are evaluated on test videosequences.

**Keywords:** digital image stabilization, parallel programming, Streaming SIMD Extensions

## Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| GCBP | – | Gray–Coded Bitplane–Matching |
| MSE | – | Mean Square Error |
| SAD | – | Sum of Absolute Deviations |
| SSE | – | Streaming SIMD Extensions |
| SIMD | – | Single Instruction, Multiple Data |

# Obsah

# Seznam obrázků

# Seznam tabulek

# Seznam výpisů zdrojového kódu

# 1 Úvod

Digitální videosekvence jsou často znehodnoceny chvěním kamery nebo snímacího zařízení. To, jak moc je videosekvence znehodnocena, závisí na vzdálenosti snímaných objektů od kamery (hloubce scény) a na stabilitě uchycení kamery. Cílem stabilizace obrazu je odstranit z videosekvence chvění způsobené nechtěným pohybem kamery. To je užitečné jednak pro zlepšení subjektivní kvality záznamu (chvění kamery působí rušivě), a také pokud se videosekvence dále automaticky zpracovává.

Stabilizátor může být buď optický nebo digitální. V případě optického stabilizátoru je pohyb kamery kompenzován pohybem čoček v objektivu nebo pohybem snímacího čipu, popřípadě nějakým gyroskopickým systémem uchycení kamery. Digitální stabilizátory kompenzují nechtěné pohyby kamery digitální úpravou snímané videosekvence. V této práci se budeme zabývat digitálními stabilizátory.

Digitální stabilizátory obrazu mají využití jednak pokud má být stabilizovaná videosekvence sledována člověkem, ale také pokud má být dále automaticky zpracovávána. Digitální stabilizátory obrazu můžeme najít nejčastěji ve videokamerách a fotoaparátech kvůli odstranění chvění způsobené pohybem kamery při natáčení [1]. Videosekvence znehodnocené chvěním kamery nejsou pro diváka příjemné, divák se také musí více soustředit a rychleji ztrácí pozornost [5]. Také například při řízení dálkově ovládaných robotů je záznam z kamery robota znehodnocen otřesy způsobenými pohybem robota po otevřeném terénu. Takový záznam však může být pro operátora obtížné sledovat a vyhodnocovat. Operátorovi je třeba ukázat pokud možno co nejplynulejší záznam z kamery robota [1, 2].

Digitální stabilizace má využití také pokud je třeba videosekvence nějak dále automaticky zpracovávat. Digitální stabilizátory se používají často jako předstupeň pro algoritmy pro automatické rozpoznávání a sledování cílů [15, 3], dále pro systémy pro řízení autonomně se pohybujících vozidel a obecně v systémech pro detekci popřípadě počítání pohybujících se objektů ve videosekvencích. Ve všech těchto aplikacích je třeba záznam z kamery před dalším zpracováním nejdříve stabilizovat, jinak by nebylo možné ve videosekvenci nalézt pohybující se objekty. Stabilizace obrazu má také využití v průmyslových čtecích zařízeních. V [19] je stabilizátor obrazu použit ke zlepšení účinnosti čtečky čárových kódů, kde čtečka je držená v ruce uživatelem.

Algoritmy pro kompresi videa také dosahují příznivějších výsledků, pokud je videosekvence před kompresí stabilizována. To je výhodné zvlášť pokud je třeba videosekvenci v reálném čase posílat po síti. Například v [8] a [1] je stabilizátor obrazu použit pro stabilizaci videohovorů.

Často je nutné stabilizovat videosekvenci v reálném čase. Pokud je navíc stabilizátor obrazu používán jako předstupeň k dalšímu automatickému zpracovávání, je třeba sekvenci nejen stabilizovat v reálném čase, ale také ponechat dostatečný výpočetní výkon, aby bylo možné další zpracování v reálném čase provádět. V [4] a [16] toho bylo dosaženo tím, že stabilizátor obrazu byl implementován na speciální hardwarové platformě, v [10] byla dokonce výpočetně nejkritičtější část algoritmu implementována hardwarově na obvodech VLSI (je však třeba dodat, že to bylo v roce 1999).

Cílem této práce je navrhnout a implementovat systém pro digitální stabilizaci videosekvencí. Systém by měl pracovat v reálném čase a pokud možno s co nejnižšími nároky na použitý hardware, tak aby byl ponechán dostatečný výpočetní výkon pro další zpracování videosekvence, například počítání pohybujících se objektů.

Práce je organizována následovně: V kapitole 2 je podán stručný přehled o existujících systémech pro stabilizaci obrazu. Funkce systémů pro stabilizaci obrazu je zde nejdříve rozdělena do dílčích úloh, které jsou pro většinu systémů společné. Jde výpočet lokálních pohybových vektorů, odhad pohybu kamery, filtrace pohybu kamery a kompenzace tohoto pohybu. V kapitole je dále ukázáno, jak existující systémy řeší tyto jednotlivé kroky. V kapitole 3 je popsána funkce systému pro stabilizaci obrazu, který byl implementován v rámci této práce. Nejdříve je opět prezentován algoritmus pro výpočet pohybových vektorů. V dalších částech kapitoly je pak ukázán postup odhadu pohybu kamery na základě těchto vektorů a také postup kompenzace tohoto pohybu. V kapitole 4 jsou identifikovány části implementovaného systému, které se ukázaly výpočetně nejnáročnější. Tyto kritické části výpočtu byly rozděleny do několika vláken, algoritmy pro výpočet lokálních pohybových vektorů byly dále optimalizovány pro využití vektorových instrukcí SSE. V kapitole 4 jsou tyto jednotlivé optimalizace popsány. V kapitole 5 je nejdříve vyhodnocena dosažená úspěšnost stabilizace obrazu. Ve druhé části kapitoly je pak prezentováno urychlení dosažené pomocí jednotlivých optimalizací a celková rychlost výpočtu programu. Poslední kapitola, kapitola 6, uzavírá tuto práci a shrnuje dosažené výsledky.

# 2 Existující metody

V této kapitole bude podán stručný přehled o existujících metodách stabilizace obrazu. Systémy pro digitální stabilizaci obrazu se obecně skládají ze dvou částí: ze systému pro určení pohybu kamery mezi dvěma snímky ve videosekvenci a ze systému pro kompenzaci tohoto pohybu. Pro určení pohybu kamery bylo vytvořeno mnoho metod. Mezi nejpoužívanější metody patří ty, které nejdříve určí tzv. lokální pohybové vektory. Zjednodušené schéma takového systému je na obrázku 1.

Na vstupu systému jsou dva po sobě jdoucí snímky z videosekvence. Snímky se rozdělí na bloky, poté se ve snímku ve videosekvenci v čase $t$ vyhledávají bloky z předchozího snímku. Zjišťuje se tak pohyb v jednotlivých částech obrazu. Vektory popisující pohyb jednotlivých bloků se nazývají *lokální pohybové vektory*. Stručný přehled o metodách pro určení lokálních pohybových vektorů je podán v kapitole 2.1.

Na základě lokálních pohybových vektorů je pak třeba určit globální pohyb kamery mezi dvěma snímky. To, jak je pohyb určen a popsán, závisí na modelu pohybu kamery, který daný systém pro stabilizaci obrazu používá. Většinou jde o posun kamery. Některé systémy podporují i afinní transformace, popř. perspektivní transformace.

Dále se filtruje pohyb kamery. Cílem této filtrace je odstranit nechtěné chvění z globálního pohybu kamery a pokud možno zachovat úmyslný pohyb kamery, jako například panorámování nebo sledování pohybujících se objektů. Tyto kroky jsou pro většinu existujících systémů pro stabilizaci obrazu společné. Algoritmy, které jednotlivé kroky řeší, se však systém od systému liší. Stručný přehled o tom, jak existující systémy řeší jednotlivé kroky stabilizace, tedy odhad pohybových vektorů, odhad pohybu kamery a kompenzaci pohybu kamery, bude podán v následujících kapitolách.

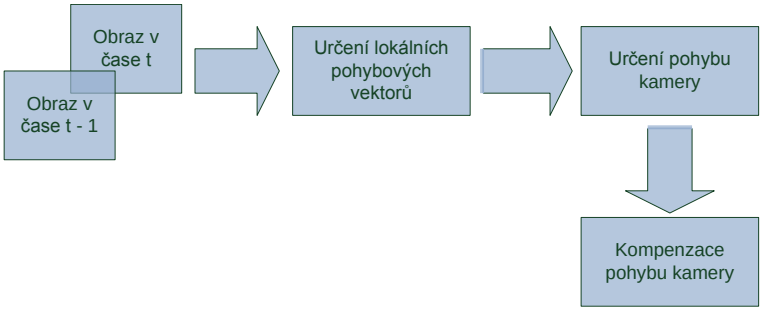
## 2.1 Odhad lokálních pohybových vektorů

V této části práce je podán přehled o jednotlivých metodách pro určení pohybových vektorů. Na základě těchto vektorů je pak určen celkový pohyb kamery. Jednotlivé algoritmy se liší podle toho, jaké snímky používají k výpočtu pohybu kamery:

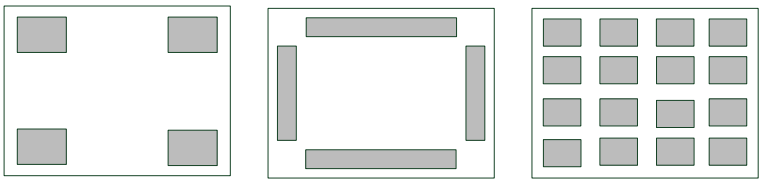
**Frame to Frame** — pohyb kamery se počítá mezi dvěma po sobě jdoucími snímky. Tento algoritmus je vhodné použít, pokud je pohyb kamery velký, popřípadě pokud kamera nesnímá stále stejné místo.

**Frame to Reference** — systém má uložen referenční snímek, pohyb kamery se počítá mezi posledním snímkem z videosekvence a tímto referenčním snímkem. Tento algoritmus je vhodné použít v případě, že je kamera namířena stále na stejné místo a jediný pohyb kamery je pohyb vzniklý nechtěnými otřesy.

Dále se budeme zabývat pouze algoritmy Frame to Frame. Určení lokálních pohybových vektorů tedy probíhá pomocí porovnávání obrazu z videosekvence s předešlým obrazem (označme jako obraz v čase $t$ a obraz v čase $t-1$). Obraz v čase $t-1$ se nejdříve rozdělí na bloky. Možnosti rozmístění jednotlivých bloků jsou znázorněny na obrázku 2. Bloky mohou být v obraze rozmístěny rovnoměrně [3, 11], často se však používá menší počet




Obrázek 1: Zjednodušené blokové schéma stabilizátoru obrazu



Obrázek 2: Možnosti uspořádání prohledávaných bloků v jednotlivých snímcích

bloků umístěných v rozích [7, 8], popřípadě po stranách obrazu [12, 13]. To proto, že předpokládáme, že v popředí se mohou vyskytovat pohybující se objekty. Naopak, v rozích a po stranách obrazu bude většinou pozadí, které by bylo na ideální sekvenci nehybné. V sekvenci znehodnocené pohybem kamery pak bude pohyb pozadí opačný vzhledem k pohybu kamery, který chceme pomocí digitálního stabilizátoru odstranit.

V obraze v čase $t$ se snažíme najít oblasti, které se co nejvíce podobají těmto blokům. Lokální pohybový vektor pro daný blok je pak určen jako rozdíl mezi souřadnicemi bloku v obraze v čase $t-1$ a souřadnicemi nalezeného bloku v obraze v čase $t$. Nejjednodušší způsob určení lokálních pohybových vektorů se nazývá *Block–Matching*. Mezi další algoritmy pro určení lokálních pohybových vektorů patří *Gray–Coded Bitplane–Matching* [9, 7] a *Laplacian Two–Bitplane Matching* [8], *Phase Correlation Matching* [12] a porovnávání zájmových bodů mezi dvěma následujícími obrazy [4]. Tyto algoritmy budou stručně popsány v následujících částech práce. Mezi další metody zjištění pohybu kamery patří metody založené na Fourier–Mellinově transformaci [5] a metody založené na waveletové transofrmaci [6].

### 2.1.1 Block matching

Nejjednodušší způsob určení lokálních pohybových vektorů se nazývá *Block–Matching*. Hledá bloky v obraze, které se nejméně liší od bloků v předcházejícím obraze ve videosekvenci. Jako kritérium odlišnosti dvou bloků můžeme zvolit součet absolutních odchylek (Sum of Absolute Differences, $SAD$) [23], popřípadě průměrnou absolutní odchylku (Mean Absolute Deviation, $MAD$) [1, 3, 11]. Součet absolutních odchylek mezi body v obraze se

vypočítá jako

$$SAD(m,n) = \sum_x \sum_y |ft(x+m, y+n) - f_{t-1}(x,y)|, \tag{1}$$

průměrná absolutní odchylka dvou bloků v obraze se vypočítá jako

$$MAD(m,n) = \frac{1}{MN} SAD(m,n). \tag{2}$$

Lokální pohybový vektor pro daný blok je takový vektor $(m,n)$, který minimalizuje $SAD(m,n)$, resp. $MAD(m,n)$. Vzhledem k tomu, že $M$ i $N$ jsou konstantní, je jedno, jestli budeme minimalizovat SAD nebo MAD.

### 2.1.2 Bitplane Matching

Počítat v každém bodě prohledávaného obrazu, resp. prohledávané oblasti, průměrnou absolutní odchylku podle vztahu (2) je značně výpočetně náročné. Proto byly vyvinuty algoritmy založené na porovnávání obrazu v jednotlivých bitových rovinách (bitplanes, odtud Biplane Matching). Tyto algoritmy nejdříve oba obrazy předzpracují a rozdělí do jednotlivých bitových rovin, bitovou rovinou se rozumí binární obraz.

Při vyhledávání pak počítají chybovou hodnotu mezi dvěma obrazy pomocí jednoduchých booleovských operací, což vede k větší efektivitě při výpočtu. Proto byl také tento algoritmus použit pro implementaci digitálního stabilizátoru obrazu v této práci. Podrobnější popis alogoritmu je možné najít v kapitole 3.1.

### 2.1.3 Pyramid Methods

Metody založené na vyhledávání podobných oblastí jsou značně výpočetně náročné [14]. Proto se kvůli optimalizaci často obraz i vzor nejdříve podvzorkují. Existují různé varianty tohoto postupu [14, 3, 15].

Podvzorkování může proběhnout v několika úrovních, tyto úrovně nazýváme stupně pyramidy. Vyhledávání pak probíhá nejdříve v obraze s nejnižším rozlišením. Poté se přejde do obrazu s vyšším rozlišením, tedy do nižší úrovně pyramidy. Zde už se neprohledává celý obraz, ale pouze okolí bodu, který odpovídá nalezené poloze vzoru v předchozím stupni pyramidy. Celý postup se opakuje, dokud nedojdeme k původnímu rozlišení, tedy k nejnižšímu stupni pyramidy.

### 2.1.4 Phase Correlation Matching

Algoritmus *Phase Correlation Matching* je založen na Fourierově transformaci. Tato metoda je použita v [12]. Mějme dva obrazy, $f_t(x,y)$ a $f_{t+1}(x,y)$, které jsou mezi sebou ve vztahu

$$f_{t+1}(x,y) = f_t(x - d_x, y - d_y).$$

Na základě vlastnosti Fourierovy transformace můžeme psát

$$F_{t+1}(u,v) = F_t(u,v) exp\left\{-j2\pi(ud_x + vd_y)\right\},$$

Obrázek 3: Ilustrace funkce algoritmu Phase Correlation Matching

kde $F_t$ značí Fourierovu transformaci $f_t$. Normalizované křížové výkonové spektrum (cross-power spectrum) signálů $f_t$ a $f_{t+1}$ je možné určit jako

$$R\,(x,y) = \frac{F_{t+1}(u,v)F_t^*(u,v)}{|F_t(u,v)F_t^*(u,v)|} = exp\left\{-j2\pi(ud_x + vd_y)\right\}, \tag{3}$$

kde $F^*(u,v)$ označuje číslo komplexně sdružené k $F(u,v)$. Jak je uvedeno v [12], posunutí $(d_x, d_y)$ je pak možné určit jako

$$(d_x, d_y) = \arg\max_{x,y}\left(r(x,y)\right),$$

kde $r$ je inverzní Fourierův obraz $R$. Ukázka funkce algoritmu je na obrázku 3. Na obrázku vlevo je původní obraz, na obrázku uprostřed je obraz posunutý. K oběma obrazům byl přidán šum, aby nebyly zcela totožné. Na obrázku vpravo je vidět výkonové spektrum $R$ vypočtené podle vztahu (3). Všimněme si, že maximum výkonového spektra, tedy nejsvětlejší bod na obrázku 3 vpravo, je bod o souřadnicích, které odpovídají posunutí obrazu uprostřed oproti původnímu obrazu.

### 2.1.5 Feature Tracking

Další ze způsobů jak určit pohybové vektory se nazývá *Feature Tracking*. Předchozí metody se pokoušely najít v novém obrazu bloky co nejpodobnější blokům z předcházejícího obrazu. Tyto bloky v předcházejícím obrazu, které byly použity jako vzory při vyhledávání, měly vždy stejnou pozici. Metody založené na Feature Trackingu, tedy vyhledávání zájmových oblastí, naproti tomu pracují ve dvou krocích: nejdříve najdou zájmové oblasti v předchozím obraze, pak se snaží tyto zájmové oblasti najít v novém obraze.

Za zájmové oblasti většinou považujeme ty oblasti obrazu, které obsahují prudké změny gradientu jasové funkce. K vyhledání zájmových oblastí existuje více způsobů, v [4, 15] se nejdříve provede konvoluce obrazu s Laplaceovou maskou, v [16] se používají waveletové masky. Obraz je nejdříve rozdělen na několik částí, podle toho kolik zájmových oblastí chceme vyhledávat. V každé části obrazu je pak vybrána jedna zájmová oblast.

Nejjednodušší způsob je použit v [15], kdy se vybere oblast okolo nejsvětlejšího bodu (v obraze získaném konvolucí s Laplaceovou maskou jde o bod na nejvýraznější hraně). Systémy prezentované v [4] a [16] používají různé heuristické přístupy k nalezení oblasti na horizontu.

Když jsou nalezeny zájmové oblasti v předchozím obraze, je třeba vyhledat odpovídající zájmové oblasti v novém obraze. To je možné provést tak, že v novém obraze procházíme okolí zájmové oblasti nalezené v minulém obraze. Snažíme se při tom opět minimalizovat rozdíl mezi vybranou oblastí a zájmovou oblastí v minulém obraze. V [4] a v [16] je k měření chyby použito SSD (Sum of Square Differences):

$$SSD(m,n) = \sum_x \sum_y \left[ f_t(x+m, y+n) - f_{t-1}(x,y) \right]^2 .$$

Vektor $(m,n)$ pro který je $SSD$ minimální, je určen jako lokální pohybový vektor.

## 2.2 Odhad globálního pohybu kamery

Na základě lokálních pohybových vektorů se určí tzv. globální pohyb kamery. Systémy pro stabilizaci obrazu je možné rozdělit podle toho, jaké druhy globálního pohybu kamery podporují. Jednotlivé systémy se zde liší tím, jaký matematický model používají k popisu pohybu kamery, resp. pohybu snímku oproti předešlému snímku. Od toho se pak odvíjí způsob výpočtu parametrů tohoto pohybu.

### 2.2.1 Posun kamery

Některé systémy uvažují pouze posun kamery podle vztahu

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 + d_x \\ y_1 + d_y \end{bmatrix},$$

kde $(x_1, y_1)$ jsou souřadnice libovolného bodu v předcházejícím snímku $(x_2, y_2)$ jsou souřadnice odpovídajícího bodu v novém snímku z videosekvence. Jediné dvě neznámé hodnoty jsou $d_x$ a $d_y$. Vektor $(d_x, d_y)$, o který je nový obraz oproti předešlému obrazu posunutý, nazýváme *globální pohybový vektor*. Složky globálního pohybového vektoru je možné určit jako modus [3] nebo medián [7, 14] složek všech lokálních pohybových vektorů. Je také možné použít nějaký složitější statistický přístup [8, 11], který v sobě zahrnuje i validaci pohybových vektorů.

### 2.2.2 Afinní transformace

Některé systémy jsou schopny na základě pohybových vektorů určit parametry pro afinní transformace [3, 15, 16, 17]. Afinní transformace ve 2D jsou obecně určeny šesti hodnotami $a_{11}$, $a_{12}$, $a_{21}$, $a_{22}$, $b_1$, $b_2$ a lze je zapsat vztahem

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}. \tag{4}$$

Afinními transformacemi lze popsat posunutí obrazu jako

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \end{bmatrix},$$

kde $(d_x, d_y)$ je vektor posunutí, a otočení obrazu kolem počátku souřadnic jako

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}, \tag{5}$$

kde $\theta$ je úhel otočení. Systémy pro stabilizaci obrazu často předpokládají pouze malou rotaci mezi dvěma po sobě následujícími snímky [3, 15]. Protože se uvažuje pouze s malým úhlem otočení a platí, že

$$\lim_{\theta \to 0} \cos\theta = 1,$$

může být $\cos\theta$ ze vztahu (5) aproximován jako 1. Pokud bude úhel $\theta$ dostatečně malý, můžeme $\sin\theta$ aproximovat lineární funkcí, resp. přímkou. Směrnice této přímky v $\theta$ pak bude $\sin'\theta = \cos\theta$. pro $\theta \to 0$ opět platí $\cos\theta \to 1$ a tím pádem $\sin\theta \approx \theta$. Toho lze využít ke zjednodušení výpočtu parametrů transformace:

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} s & \theta \\ -\theta & s \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \end{bmatrix}. \tag{6}$$

Tento zjednodušený model je použit v systému prezentovaném v [15]. Pokud do vztahu (6) dosadíme postupně za $(x_1, y_1)$ počáteční body a za $(x_2, y_2)$ koncové body všech lokálních pohybových vektorů, dostaneme předeterminovanou soustavu rovnic. Řešením této soustavy jsou pak parametry transformace $\theta$, $d_x$, $d_y$ a $s$. Systém prezentovaný v [3] používá k určení parametrů transforace jiný zjednodušený model:

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_1 - \alpha \\ y_1 - \beta \end{bmatrix} + \begin{bmatrix} \alpha \\ \beta \end{bmatrix},$$

kde $\alpha$ a $\beta$ je možné vypočítat při znalosti středu otočení, úhlu otočení a vektoru posunutí. Opět, pokud dosadíme za $(x_1, y_1)$ počáteční body a za $(x_2, y_2)$ koncové body všech lokálních pohybových vektorů, dostaneme předeterminovanu soustavu rovnic, jejíž řešením jsou parametry transformace. Tento model byl použit v implementovaném systému a bude blíže popsán v kapitole 3.2.4. Systém prezentovaný v [17] používá mírně odlišný přístup, snaží se odhadnout všechny parametry afinní transformace uvedené ve vztahu (4), a to pomocí minimalizace chybové funkce

$$E_t(\overrightarrow{m}) = \sum_x \sum_y \left[ f_t(x,y) - f_{t-1}(a_{11}x + a_{12}y + d_x, a_{21}x + a_{22}y + d_y) \right]^2,$$

kde $\overrightarrow{m} = (a_{11}, a_{12}, a_{21}, a_{22}, b_1, b_2)$. V [17] je pak uveden postup, jak tuto funkci minimalizovat.

### 2.2.3 Validace pohybových vektorů

Lokální pohybové vektory ne vždy popisují pohyb pozadí, který chceme kompenzovat. Pokud se ve videosekvenci vyskytují pohybující se objekty, mohou některé lokální pohybové vektory zachycovat pohyb těchto objektů. Metody odhadu lokálních pohybových vektorů navíc nejsou stoprocentně spolehlivé, například v případech, kdy bloky v obraze použité k určení jednotlivých pohybových vektorů neobsahují žádné náhlé změny jasu, obsahují nějaký periodický vzor, jsou silně zatíženy šumem, atd.

Proto se používají různé metody validace pohybových vektorů [11, 14, 8]. Metody pro validaci vektorů je možné rozdělit do dvou skupin, a to na metody používající různé statistické metody k vyřazení chybných pohybových vektorů a na metody používající Kalmanův filtr.

**2.2.3.1 Statistické metody** Cílem těchto metod je vyřadit ze zpracování bloky, které vyhovují některé z následujících podmínek:

**Obsahují velké plochy stejné barvy** — Takové bloky neobsahují žádné hrany, všechny jejich body mají téměř stejnou barvu. Většina algoritmů pro párování bloků v takových podmínkách nefunguje dobře [14]. V [14] a [11] jsou za takové bloky považovány ty, které mají nízkou směrodatnou odchylku jasových hodnot v jednotlivých pixelech.

**Mají nízký odstup signálu od šumu** — Jde o oblasti, kde je zachycena např. tráva, listí a podobné objekty. V [14] jsou za takové bloky považovány ty, pro které je hodnota chybového operátoru použitého pro párování bloků (MAD u Block Matchingu, například) větší, než experimentálně stanovená prahová hodnota.

**Obsahují opakující se vzor** — V oblastech, kde se periodicky opakuje nějaký vzor, může algoritmus pro párování bloků vybrat kterýkoliv z prvků opakujícího se vzoru. V [14] jsou takově bloky identifikovány následovně: Nechť $min_1(\text{MAD})$ a $min_2(\text{MAD})$ jsou nejmenší, resp. druhá nejmenší hodnota chybového operátoru použitého pro párování bloků, a $min_1(\text{MV})$ a $min_2(\text{MV})$ jsou odpovídající pohybové vektory. V [14] je jako oblast s periodicky se opakujícím vzorem určena oblast, pro kterou platí

$$\begin{aligned} \frac{min_1(\text{MAD})}{min_2(\text{MAD})} &> t_1, \qquad \text{a zároveň} \\ |min_1(\text{MV}) - min_2(\text{MV})| &> t_2. \end{aligned}$$

Hodnoty $t_1$ a $t_2$ jsou zde experimentálně určené prahové hodnoty.

**2.2.3.2 Kalmanův Filtr** Jiné systémy, například [1, 12, 13], používají k validaci pohybových vektorů Kalmanův filtr. Kalmanův filtr je obecně určen k odhadu stavu diskrétního systému na základě znalosti dynamiky jeho chování a jeho předchozího stavu. V oblasti stabilizace obrazu to znamená, že pomocí Kalmanova filtru můžeme odhadnout pohyb kamery v nějakém čase $t$, pokud známe pohyb kamery v čase $t-1$ a dynamický model

chování tohoto pohybu. Model dynamiky systému (v našem případě pohybu kamery) je popsán jako

$$x(t+1) = Fx(t) + w(t),$$

kde $w(t)$ je šum vznikající v systému. Do filtru vstupují pozorování stavových proměnných definované jako

$$y(t) = Hx(t) + v(t),$$

kde $v(t)$ je opět šum způsobený chybami pozorování, například některými z chyb zmíněných v odstavci 2.2.3.1. Předpokládá se, že šum vznikající v systému a šum vznikající při pozorování jsou na sobě nezávislé a mají normální rozdělení s nulovou střední hodnotou.

V [12] a [1] jsou uvedeny dva modely pohybu kamery: CAMM, *Constant Acceleration Motion Model* a CVMM, *Constant Velocity Motion Model*. Chvění kamery je oproti frekvenci pořizování snímků ve videosekvenci relativně pomalé [12]. Proto by globální pohybový vektor pro nový snímek měl být podobný jako ten z předchozího snímku. Při stabilizaci obrazu pomocí Kalmanova filtru nejdříve pomocí filtru odhadneme pohyb kamery. Lokální pohybové vektory, které se příliš liší od vektoru odhadnutého Kalmanovým filtrem pak můžeme označit za chybné a z dalšího zpracování je vyřadit.

## 2.3 Kompenzace pohybu kamery

Finální část procesu stabilizace je kompenzace nechtěného pohybu kamery. Z videosekvence chceme odstranit nechtěné chvění kamery, zatímco úmyslný pohyb kamery, jako například sledování nějakého pohybujícího se objektu nebo panorámování, chceme zachovat.

V této fázi už máme k dispozici informaci o globálním pohybu kamery, buď o jejím posunu, nebo otočení, nebo o pohybu popsaném nějakou komplexnější transformací. Záleží, jaký model pohybu kamery používáme. Nyní jde o to, jak tento pohyb filtrovat, aby byl co nejplynulejší. Než přistoupíme k stručnému přehledu jednotlivých přístupů k tomu, jak toho dosáhnout, je třeba systémy rozdělit na dvě kategorie:

**Systémy s pamětí transformace** — tyto systémy počítají pohyb kamery mezi dvěma po sobě jdoucími *nestabilizovanými* snímky. Musí si proto uchovávat paměť o transformaci, kterou se snímky provádí.

**Systémy bez paměti transformace** — tyto systémy nemají žádnou vnitřní paměť transformace, resp. stavu ve kterém se nacházejí. Pohyb kamery vždy počítají oproti předcházejícímu *stabilizovanému* snímku.

Dále se budeme zabývat pouze systémy s pamětí transformace. Nejjednodušší přístup, jak odfiltrovat neúmyslný pohyb kamery, je použitý v [11]. Tento systém kompenzuje pouze posun kamery. Pokud je některá ze složek $(d_x, d_y)$ větší než předdefinovaná prahová hodnota, pohyb kamery je považován za úmyslný a žádná kompenzace se neprovádí. V opačném případě se snímek posune o vektor opačný k vektoru popisujícímu pohyb kamery.

V [9] a [7] je použit jednoduchý filtr, který je určen k tomu, aby potlačil nechtěné chvění kamery. Tento systém kompenzuje opět pouze posun kamery. Vektor $V_a^t$, který bude kompenzován (v [9] je označen jako integrovaný pohybový vektor) je zde určen jako

$$V_a^t = DV_a^{t-1} + V_g^t, \tag{7}$$

kde $V_a^{t-1}$ je integrovaný pohybový vektor v čase $t-1$ a $V_g^t$ je globální pohybový vektor určující pohyb kamery mezi snímkem v čase $t$ a snímkem v čase $t-1$. $D$ je koeficient, který zajišťuje, že pokud se kamera přestane pohybovat, $V_a^t$ konverguje k 0, $0 < D < 1$. Tím se zajistí zachování úmyslného pohybu kamery. Rovnici (7) je možné chápat také jako IIR filtr prvního řádu. V [18] je ke stejnému účelu použit FIR filtr čtvrtého řádu. Kalmanův filtr použitý v [1], [12] a [13] slouží kromě validace vektorů také k filtraci pohybu.

Poslední částí kompenzace pohybu kamery je pak posunutí, resp. otočení obrazu inverzní k vypočteným hodnotám pohybu kamery. V případě posunutí pouze stačí obraz posunout o vektor opačný k vektoru pohybu kamery. V případě otočení je třeba použít nějakou metodu interpolace, většinou bilineární interpolaci [3, 16, 17]. Jinak by došlo ke znehodnocení obrazu vlivem zaokrouhlování hodnot souřadnic jednotlivých pixelů při otáčení.

# 3 Popis navrhovaného řešení

V této kapitole je prezentován systém pro digitální stabilizaci obrazu, který byl implementován v rámci této práce. Blokové schéma systému je na obrázku 4. Do systému vstupují postupně snímky z videosekvence. V každém snímku jsou nejdříve nalezeny lokální pohybové vektory. Tyto vektory jsou pak rozděleny na ty, které odpovídají pozadí, a na ty, které odpovídají pohybujícím se objektům. Vektory, které odpovídají pohybujícím se objektům, jsou z dalšího zpracování vyřazeny. U zbylých vektorů se určí, zda popisují posun nebo rotaci kamery. Dále se odhadnou parametry posunu, resp. rotace a tato transformace se zahrne do paměti systému (systém si uchovává transformaci, kterou s jednotlivými snímky provádí). Nakonec dojde k odpovídajícímu posunutí, resp. otočení obrazu. Všechny moduly funkce systému budou postupně popsány v následujících kapitolách.

## 3.1 Určení lokálních pohybových vektorů

V každém snímku jsou nejdříve nalezeny lokální pohybové vektory. Snímek je vždy rozdělen na jednotlivé bloky, ve kterých probíhá hledání pohybových vektorů. Bloky jsou rozmístěny rovnoměrně po celém snímku, jako na obrázku 2 vpravo. Počet řad a sloupců, ve kterých budou bloky rozmístěny, je možné nastavit. Systém rozmísťuje bloky tak, aby byly pokud možno co nejdál od sebe. V případě, že jsou zvoleny dvě řady a dva sloupce bloků, jsou bloky umístěny do rohů snímku jako na obrázku 2 vlevo.

V každém z těchto bloků pak vyhledáváme oblast, která se nejvíce podobá oblasti odpovídající tomuto bloku na předchozím snímku. K jejímu nalezení jsme vybrali algoritmus *Gray Coded Bitplane Matching*. Tento algoritmus byl popsán v [9]; jde o jeden z nejefektivnějších způsobů určení lokálních pohybových vektorů [1, 7, 3, 8, 9]. Algoritmus pracuje s obrazy ve stupních šedi. Nejdříve předzpracuje oba vstupní obrazy, výsledkem předzpracování jsou tzv. bitové roviny (bitplanes). V bitových rovinách se pak provádí párování bloků obvyklým způsobem, tj. počítá se chybová hodnota pro různá umístění daného bloku v prohledávaném obraze a hledá se umístění s nejmenší chybovou hodnotou. Odlišnost tohoto algoritmu od klasického block-matchingu spočívá v právě počítání oné chybové hodnoty - k výpočtu se používá pouze operace XOR, což by mělo být rychlejší než odčítání a počítání absolutní hodnoty. Algoritmus bude podrobněji popsán v následujících odstavcích.

Označme bod obrazu z videosekvence o souřadnicích $x, y$ v čase $t$ jako

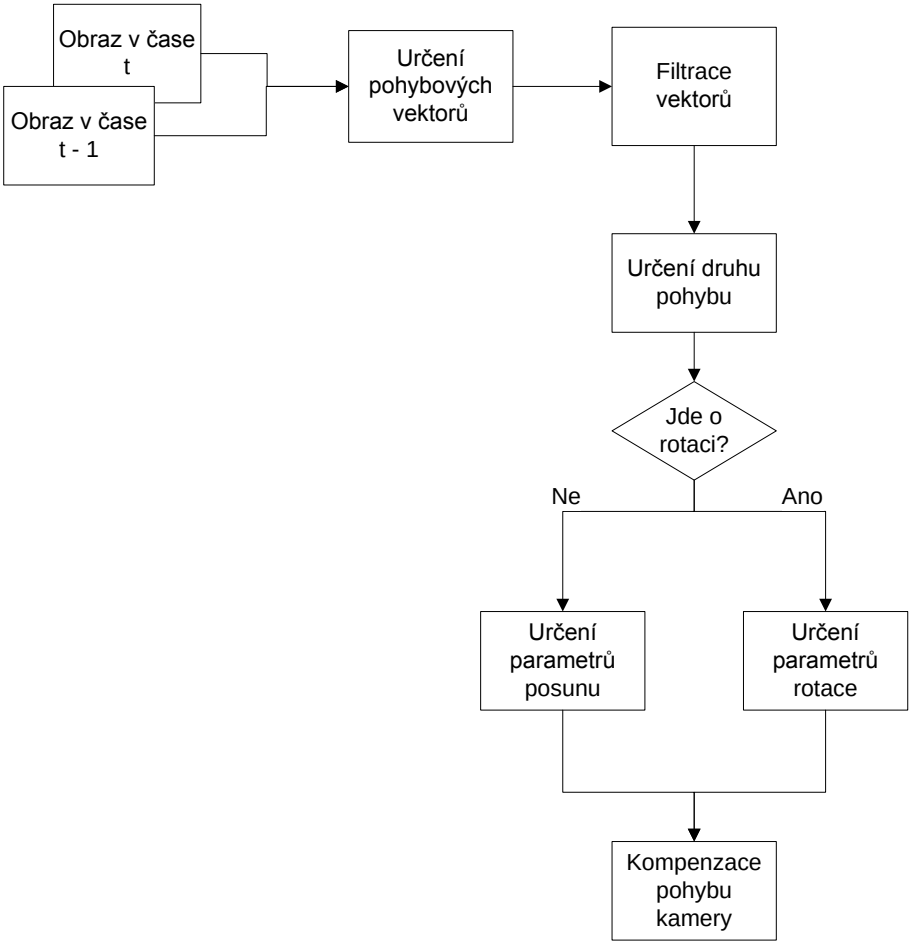
$$f_t(x, y) = a_{K-1}2^{K-1} + a_{K-2}2^{K-2} + \cdots + a_0 2^0, \tag{8}$$

kde $K$ je počet bitů použitých k reprezentaci obrazu a $a_{K-1}, a_{K-2}, \ldots, a_0$ jsou hodnoty jednotlivých bitů v této reprezentaci od nejvýznamnějšího k nejméně významnému bitu. Dále označme hodnoty bitů v jednotlivých bitových rovinách jako

$$b_{K-1}, b_{K-2}, \ldots, b_0.$$

Jak je ukázáno v [7], hodnoty bitů v jednotlivých bitových rovinách určíme jako

$$\begin{aligned} b_{K-1} &= a_{K-1} \\ b_k &= a_k \oplus a_{k+1} \quad \text{pro } 0 \leq k \leq K-2, \end{aligned} \tag{9}$$




Obrázek 4: Blokové schéma implementovaného systému

Obrázek 5: Původní snímek použitý k výpočtu bitových rovin na obrázku 6

kde $\oplus$ označuje operátor nonekvivalence, tedy bitové XOR. Na obrázku 6 jsou ukázány jednotlivé bitové roviny obrázku 5 vypočtené podle vztahu (9). Všimněme si, že nejvíce vizuálních informací obsahují první čtyři bitové roviny $b_7$–$b_4$. Obsah zbylých čtyř bitových rovin je možné považovat za šum.

V [9], [10], [8] a [7] se k odhadu lokálních pohybových vektorů vždy používá jen jedna bitová rovina. V [9] je to šestá bitová rovina, v [7] pátá. Hodnota udávající, jak moc se porovnávané obrazy liší, tedy chybová hodnota, kterou se snažíme minimalizovat, abychom našli obraz, který se co nejvíce podobá původnímu obrazu, se vypočítá jako

$$\epsilon_k(m,n) = \sum_x \sum_y b_{k_t}(x,y) \oplus b_{k_{t-1}}(x+m, y+n), \tag{10}$$

kde $k$ je číslo bitové roviny, $\oplus$ značí bitové XOR, a $M \times N$ jsou rozměry prohledávané oblasti v pixelech. Hodnoty $m, n$, pro které je $\epsilon(m,n)$ minimální, jsou pak souřadnice hledaného lokálního pohybového vektoru.

Stabilizátor obrazu implementovaný v rámci této práce používá všech osm bitových rovin. Vstupní obrazy mají osmibitovou hloubku, hodnoty jednotlivých bitů popisuje vztah (8).

Jak je vidět na obrázku 7 vlevo, hodnoty bitů ve všech osmi bitových rovinách je možné získat najednou jako

$$a \oplus (a >> 1) = b,$$

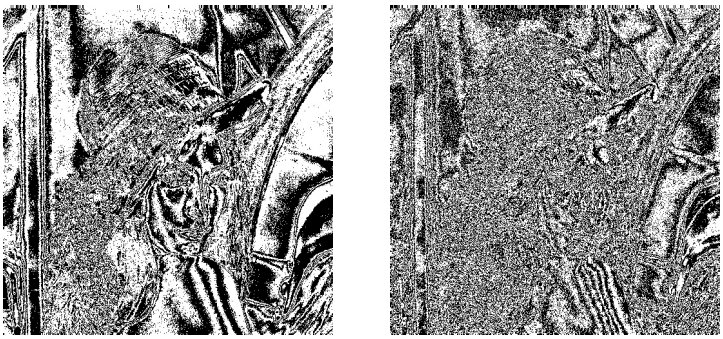
kde $a$ je hodnota jasu pixelu v původním obraze, operátor $>>$ označuje bitový posun doprava a

$$b = b_{K-1}2^{K-1} + b_{K-2}2^{K-2} + \cdots + b_0 2^0$$

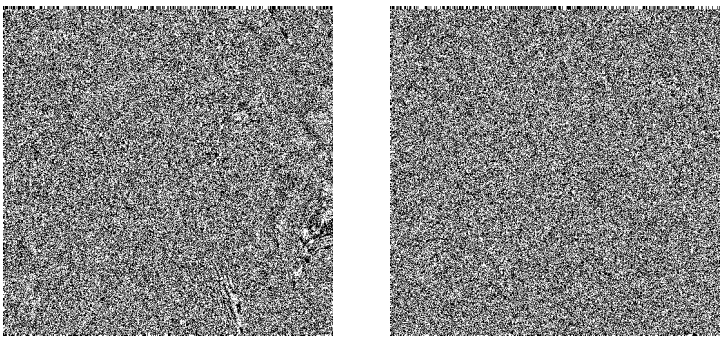
je osmibitové číslo, jehož jednotlivé bity odpovídají hodnotám bitových rovin v daném bodě obrazu. Podařilo se nám zde zakódovat všechny bitové roviny najednou pomocí dvou operací: jedné operace XOR a jedné operace bitového posunu. Pokud budeme hodnoty $b$

(a) $b_7$ (b) $b_6$

(c) $b_5$ (d) $b_4$

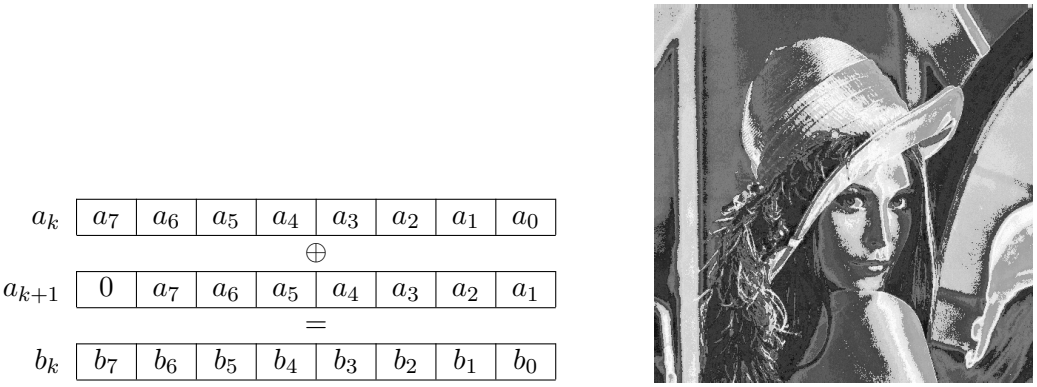


(e) $b_3$ (f) $b_2$



(g) $b_1$ (h) $b_0$

Obrázek 6: Jednotlivé bitové roviny obrázku 5




Obrázek 7: Kódování bitových rovin

považovat opět za hodnoty jasu, můžeme se na výsledný obraz podívat: je na obrázku 7 vpravo. Chybová hodnota pro všech osm bitových rovin se pak vypočítá jako

$$E(m,n) = \sum_x \sum_y \left( \epsilon(m,n)_{K-1} 2^{K-1} + \epsilon(m,n)_{K-2} 2^{K-2} + \cdots + \epsilon(m,n)_0 2^0 \right) = \sum_x \sum_y b_t(x,y) \oplus b_{t-1}(x+m, y+n). \quad (11)$$

Algoritmus používá téměř výhradně operace XOR a operace bitového posunu. Tato vlastnost algoritmu umožňuje jednak efektivní hardwarovou implementaci [7, 9], ale také efektivní paralelní implementaci v jazyce symbolických instrukcí na procesorech schopných vykonávat instrukce SSE2, jak bude ukázáno v následujících kapitolách.

## 3.2 Určení pohybu kamery

Po nalezení lokálních pohybových vektorů je třeba na základě těchto vektorů určit globální pohyb kamery. Náš systém je schopen rozlišit dva druhy pohybu kamery: posunutí a rotaci. Modul po určení pohybu kamery pracuje takto:

- Odfiltruje pohybové vektory zachycující pohyb objektů ve videosekvenci (a ne pohyb pozadí) pomocí algoritmu Mean Shift.
- Poté analyzuje zbylé vektory aby zjistil, jestli jde o posunutí nebo o rotační pohyb.
- Vypočítá hodnotu posunutí, popřípadě střed a úhel rotace.

### 3.2.1 Mean Shift

Obecnou vadou systémů pro stabilizaci obrazu založených na lokálních pohybových vektorech je fakt, že jsou citlivé na pohybující se objekty v obraze. Lokální pohybové vektory nemusí nutně odrážet pohyb kamery. Pokud je ve videosekvenci nějaký pohybující se objekt, některé lokální pohybové vektory mohou odrážet pohyb tohoto objektu, což znehodnotí

odhad celkového pohybu kamery. Navíc nemáme žádné *a priorní* informace o počtu pohybujících se objektů ve videosekvenci. Bylo by tedy dobré nějak oddělit pohybové vektory, které zachycují pohyb kamery od těch, které zachycují pohyb objektů ve videosekvenci. K filtraci pohybových vektorů je v našem systému použita populární shlukovací metoda *Mean Shift* [20, 21, 22]. Metoda nevyžaduje žádnou *a priorní* znalost výsledného počtu shluků ani tvaru jednotlivých shluků.

Na vstupu algoritmu jsou jednotlivé pohybové vektory $\boldsymbol{x_i} = (x_i, y_i), i = 1, 2, \ldots, N$. Algoritmus je obecně schopen pracovat v $d$-rozměrném prostoru, v našem případě jde o dvourozměrnou variantu. Odhad hustoty rozložení pohybových vektorů v rovině získáme podle předpisu

$$\tilde{f}(\boldsymbol{x}) = \frac{1}{nh^d} \sum_{i=1}^{N} K\left(\frac{\boldsymbol{x} - \boldsymbol{x_i}}{h}\right),$$

kde kernel $K(x)$ je definován jako

$$K(x) = c_{k,d} k(\left\| x^2 \right\|)$$

Parametr $h$ reprezentuje poloměr okna a $c_{k,d}$ je normalizační konstanta závislá na volbě funkce $k(x)$, která je označována jako profil kernelu $K$. Snažíme se nalézt lokální maxima hustoty $\tilde{f}$, která se nacházejí v nulových bodech jejího gradientu. Vzhledem k tomu, že platí $\nabla \tilde{f} = \tilde{\nabla} f$, můžeme hledaný odhad gradientu funkce získat z gradientu odhadu funkce takto:

$$\begin{aligned} \nabla \tilde{f}(\boldsymbol{x}) &= \frac{1}{nh^d} \sum_{i=1}^{N} \nabla \left( c_{k,d} k \left( \left\| \frac{\boldsymbol{x} - \boldsymbol{x_i}}{h} \right\|^2 \right) \right) \\ &= \ldots \\ &= \frac{2c_{k,d}}{nh^{d+2}} \sum_{i=1}^{N} (\boldsymbol{x} - \boldsymbol{x_i})\, k \left( \left\| \frac{\boldsymbol{x} - \boldsymbol{x_i}}{h} \right\|^2 \right). \end{aligned} \tag{12}$$

Dále označme $g(z) = -k'(z)$, kde profil $k$ má tvar

$$k(z) = \begin{cases} 1 - z & \text{pro } z \leq 1 \\ 0 & \text{jinak} \end{cases}$$

a $z = \|x\|$. Po dosazení do (12) pak dostaneme

$$\begin{aligned} \nabla \tilde{f}(\boldsymbol{x}) &= \frac{2c_{k,d}}{nh^{d+2}} \sum_{i=1}^{N} (\boldsymbol{x_i} - \boldsymbol{x})\, g \left( \left\| \frac{\boldsymbol{x} - \boldsymbol{x_i}}{h} \right\|^2 \right) \\ &= \frac{2c_{k,d}}{nh^{d+2}} \left[ \sum_{i=1}^{N} g \left( \left\| \frac{\boldsymbol{x} - \boldsymbol{x_i}}{h} \right\|^2 \right) \right] \cdot \left[ \frac{\sum_{i=1}^{N} \boldsymbol{x_i} g \left( \left\| \frac{\boldsymbol{x} - \boldsymbol{x_i}}{h} \right\|^2 \right)}{\sum_{i=1}^{N} g \left( \left\| \frac{\boldsymbol{x} - \boldsymbol{x_i}}{h} \right\|^2 \right)} - \boldsymbol{x} \right]. \end{aligned}$$

Výraz

$$m_h(\boldsymbol{x}) = \frac{\sum_{i=1}^{N} \boldsymbol{x_i} g \left( \left\| \frac{\boldsymbol{x} - \boldsymbol{x_i}}{h} \right\|^2 \right)}{\sum_{i=1}^{N} g \left( \left\| \frac{\boldsymbol{x} - \boldsymbol{x_i}}{h} \right\|^2 \right)} - \boldsymbol{x}$$

Obrázek 8: Filtrace lokálních pohybových vektorů pomocí algoritmu Mean–Shift

se označuje jako vektor *Mean Shift* v bodě $x$. Vektor mean shift vždy směřuje směrem, ve kterém je největší nárůst odhadu hustoty $\tilde{f}$. Pomocí tohoto vektoru nyní můžeme pro každý bod určit lokální maximum hustoty, a to tak, že:

- Jako výchozí bod určíme pod o souřadnicích daného lokálního pohybového vektoru, $\boldsymbol{x_i^0} = \boldsymbol{x_i}$.
- Postupně posouváme kernel směrem největšího nárůstu hustoty, $\boldsymbol{x_i^{t+1}} = \boldsymbol{x_i^t} + m_h(\boldsymbol{x})$
- Celý proces opakujeme, dokud velikost vektoru $m_h(\boldsymbol{x})$ není nulová, resp. dostatečně malá k tomu, abychom mohli posouvání považovat za ukončené.

Lokální pohybové vektory, pro které procedura Mean Shift dospěje do stejného lokálního maxima, jsou pak zařazeny do stejného shluku. Systém dále pracuje jen s těmi lokálními pohybovými vektory, které náleží k největšímu shluku (shluku s největším počtem bodů).

Ukázka funkce algoritmu Mean–Shift je na obrázku 8. Jde o snímek z jedné z testovacích videosekvencí. Na této videosekvenci jsou zachycena pohybující se auta. Vzhledem k tomu, že auta zabírají poměrně velkou část obrazu, hodně pohybových vektorů popisuje právě pohyb jedoucího auta a ne pohyb kamery. Na obrázku 8 jsou barevně označeny bloky, které byly nalezeny jako bloky nejpodobnější blokům z minulého snímku, černě jsou označeny pozice bloků minulého snímku, které sloužily jako základ pro vyhledávání. Rozdíl v poloze barevného bloku a odpovídajícího černého bloku je tedy pohybový vektor příslušný k danému bloku. Zeleně jsou pak označeny bloky, které algoritmus Mean-Shift zařadil do největšího shluku, ostatní bloky jsou označeny červeně. Všimněme si, že největší shluk zde tvoří vektory popisující pohyb pozadí. Vektory popisující pohyb auta byly algoritmem zařazeny do jiného shluku, a tedy vyřazeny z dalšího zpracovávání.

### 3.2.2 Určení druhu pohybu kamery

Tento modul rozhoduje, jestli pohybové vektory popisují posun nebo rotaci kamery. Rozhoduje se na základě rozptylu jednotlivých složek pohybových vektorů. Rozptyl je definován jako střední hodnota kvadrátů odchylek od střední hodnoty:

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{N}\left(x_i - E\left(x\right)\right)^2.$$

Pokud je rozptyl $x$-ových i $y$-ových složek pohybových vektorů větší než předdefinovaná prahová hodnota, pohyb se považuje za rotaci. V opačném případě je pohyb považován za posunutí.

### 3.2.3 Posun kamery

Pokud je pohyb kamery klasifikován jako posun, vypočtou se složky globálního pohybového vektoru jako mediány složek jednotlivých lokálních pohybových vektorů.

### 3.2.4 Rotace kamery

Jak je uvedeno v [3], posunutí a rotace (nebo pouze rotace) obrazu může být popsáno následujícím zjednodušeným modelem:

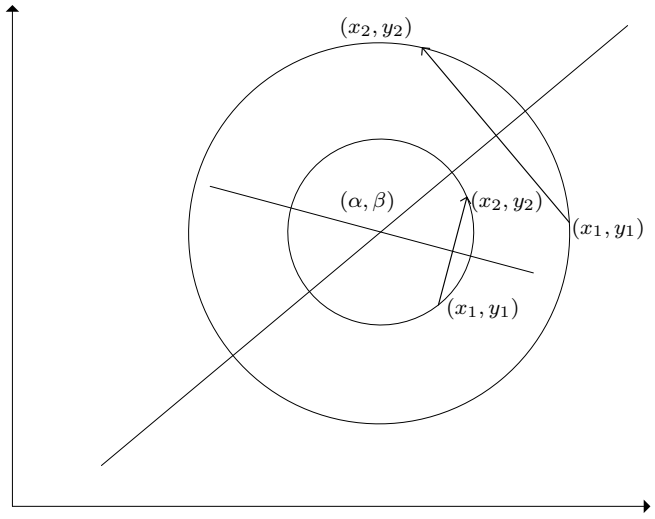
$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_1 - \alpha \\ y_1 - \beta \end{bmatrix} + \begin{bmatrix} \alpha \\ \beta \end{bmatrix},$$

$\alpha$ a $\beta$ jsou v [3] definovány jako

$$\begin{aligned} \alpha = {} & \frac{\cos\theta \cdot x_0 + \sin\theta \cdot y_0 + x_0 + d_x}{2} + \\ & + \frac{\sin\theta(\sin\theta \cdot x_0 + \cos\theta \cdot y_0 - y_0 - d_y)}{2(1-\cos\theta)}, \end{aligned}$$

$$\begin{aligned} \beta = {} & \frac{1}{1-\cos\theta}[-\sin\theta \cdot x_0 - \cos\theta \cdot y_0 + y_0 + d_y + \\ & + \sin\theta \cdot \frac{\cos\theta \cdot x_0 + \sin\theta \cdot y_0 + x_0 + d_x}{2} + \\ & + \sin^2\theta \cdot \frac{\sin\theta \cdot x_0 + \cos\theta \cdot y_0 - y_0 - d_y}{2(1-\cos\theta)}], \end{aligned}$$

kde $(x_0, y_0)$ je střed rotace, $\theta$ je úhel rotace ($\theta \neq n\pi$) a $(d_x, d_y)$ je vektor posunutí. Na základě znalosti lokálních pohybových vektorů můžeme odhadnout parametry pohybu $(x_0, y_0, \theta)$. Nejprve určíme střed otočení. Střed otočení bude ležet v průsečíku os jednotlivých pohybových vektorů, viz obrázek 9. Rovnice popisující osy jednotlivých vektorů je možné určit na základě souřadnic těchto vektorů, průsečík těchto os pak získáme řešením soustavy rovnic, ve které každá rovnice popisuje osu jednoho vektoru. Když známe střed




Obrázek 9: Odhad středu rotace

otočení, úhel otočení můžeme opět určit z pohybových vektorů pomocí goniometrických funkcí. V následujících odstavcích bude postup určení středu a úhlu popsán detailněji.

Nejprve označme složky lokálních pohybových vektorů jako

$$(u = x_2 - x_1, v = y_2 - y_1)$$

kde $(x_2, y_2)$ jsou souřadnice pixelu v obrazu v čase $t$ a $(x_1, y_1)$ jsou souřadnice odpovídajícího pixelu v předešlém obrazu a $(u,\ v)$ jsou složky pohybového vektoru. Osu každého pohybového vektoru můžeme vyjádřit jako

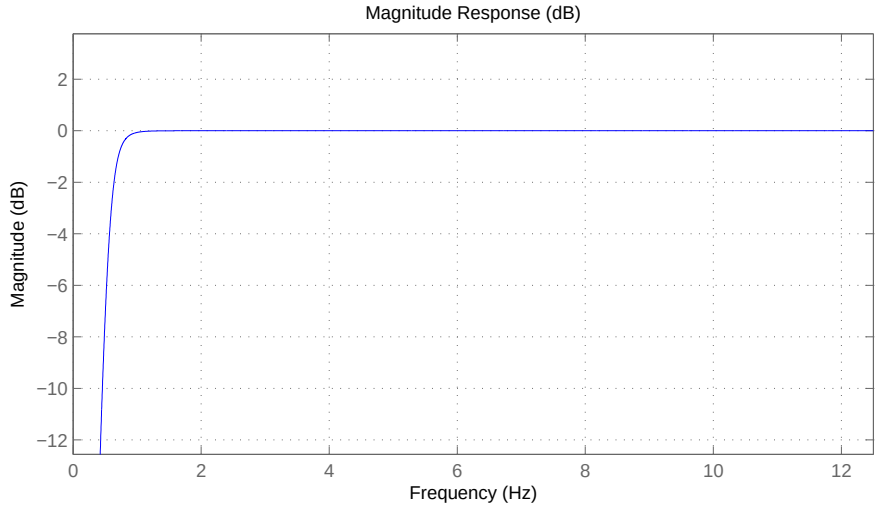
$$y = ax + b,$$

kde

$$\begin{aligned} a &= -\frac{u}{v} \\ b &= \left(y + \frac{v}{2}\right) - a\left(x + \frac{u}{2}\right). \end{aligned}$$

Jak je vidět na obrázku 9, střed otáčení bude ležet v průsečíku os pohybových vektorů. Pokud definujeme soustavu rovnic $Ax = b$ vztahem

$$\begin{bmatrix} -a_1 & 1 \\ -a_2 & 1 \\ \vdots & \vdots \\ -a_N & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix},$$




Obrázek 10: Frekevenční charakteristika filtru použitého pro filtraci pohybu kamery

řešením této soustavy bude střed otočení o souřadnicích $(\alpha, \beta)$. Úhel otočení se pak vypočítá jako

$$\theta = \tan^{-1} \frac{(y_2 - \beta)(x_1 - \alpha) - (x_2 - \alpha)(y_1 - \beta)}{(x_2 - \alpha)(x_1 - \alpha) + (y_2 - \beta)(y_1 - \beta)}$$

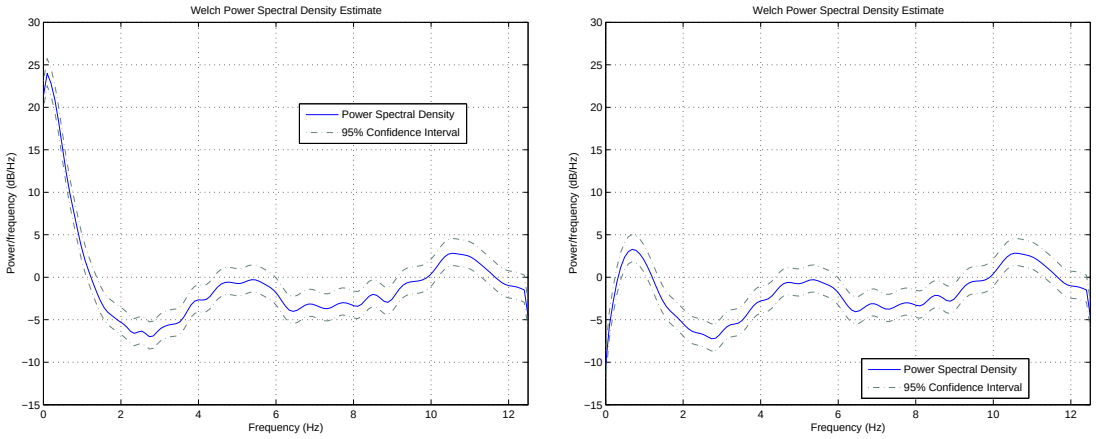
## 3.3 Kompenzace pohybu kamery

Z pohybu kamery je třeba odfiltrovat pouze nežádoucí chvění kamery, plynulé pohyby kamery je třeba zachovat. Proto jsou nejdříve složky pohybu kamery filtrovány filtrem typu horní propust. Tento filtr propustí pouze vysokofrekvenční složky pohybu kamery, tedy nechtěné chvění. Ty budou pak dále z videosekvence odstraněny. Naopak nízkofrekvenční složky pohybu, tedy plynulé otáčení kamerou, zůstane ve videosekvenci zachováno. Pro konkrétní implementaci v systému byl zvolen filtr s nekonečnou impulzní odezvou, který signál filtruje podle vztahu

$$y(n) = \sum_{i=0}^{Q} b_i x\,(n-i) - \sum_{j=1}^{Q} a_j y\,(n-j), \tag{13}$$

kde $Q$ je řád filtru $x(n)$ a $y(n)$ je vstup, resp. výstup v čase $n$ a $b_0, b_1 \ldots, b_Q, a_1, a_2 \ldots, a_Q$ jsou koeficienty filtru. Filtr byl navržen v prostředí MATLAB tak, aby tlumil frekvence nižší než 1Hz, k návrhu byla použita Čebyševova aproximace druhého druhu. Koeficienty filtru navržené MATLABem pak byly v programu dosazeny do vztahu (13). Frekvenční charakteristika filtru je na obrázku 10.

Na obrázku 11 vlevo je frekvenční spektrum horizontální složky pohybu kamery pro jednu z ukázkových videosekvencí. Jde o sekvenci zachycenou kamerou umístěnou na čelním skle jedoucího auta. Vidíme, že spektrum pohybu kamery obsahuje hodně nízkých




Obrázek 11: Frekvenční spektrum horizontální složky pohybu kamery před a po filtraci

frekvencí, menších než cca 1Hz, a dále pak vyšší frekvence. Předpokládáme, že pohyb o vyšších frekvencích je nechtěné chvění kamery. Tento pohyb tedy filtr propustí, aby mohl být později z videosekvence odstraněn. To je vidět na obrázku 11 vpravo - frekvence vyšší než 1Hz zůstaly filtrem nezměněny. Pohyb o nízkých frekvencích je naopak považován za úmyslný, konkrétně v tomto případě jde o zatáčení auta. Filtr ho tedy potlačí (opět viz obrázek 11 vpravo) a zbylé části systému se jím nebudou dále zabývat. Tento pohyb tedy zůstane ve videosekvenci zachován. Stejným způsobem je na vstupu do tohoto modulu filtrován i vertikální pohyb a rotace kamery.

Systém si uchovává informaci o stavu, ve kterém se nachází, v transformační matici $3 \times 3$. Každý nový odhadnutý pohyb kamery je nejdříve filtrován podle postupu uvedeného výše a pak je vyjádřen také transformační maticí. Pro posun o vektor $(d_x, d_y)$ má transformační matice tvar

$$T = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix}, \tag{14}$$

pro otočení o úhel $\alpha$ okolo bodu o souřadnicích $(c_x, c_y)$

$$T = \begin{bmatrix} \cos\alpha & \sin\alpha & c_x\left(1-\cos\alpha\right) - c_y \sin\alpha \\ -\sin\alpha & \cos\alpha & c_x \sin\alpha - c_y\left(1-\cos\alpha\right) \\ 0 & 0 & 1 \end{bmatrix}. \tag{15}$$

Jednotlivé transformace prováděné s obrazy ve videosekvenci je třeba skládat, jinak by nebylo možné sekvenci stabilizovat. Skládání transformací se v programu provádí násobením transformačních matic. Systém má uloženou matici transformace, při každém snímku je tato matice zleva vynásobena maticí transformace odpovídající vypočítanému posunutí, resp. otočení daného snímku oproti předchozímu snímku.

Při velkém pohybu kamery by však skládání jednotlivých pohybů mohlo způsobit, že výsledný snímek vzniklý posunutím resp. otočením opačným k tomuto pohybu by mohl skončit z větší části mimo zobrazovanou oblast. Je proto nutné, aby při nulovém pohybu

kamery matice transformace, kterou má systém uloženou, zkonvergovala k jednotkové matici.

Systém má uloženou matici, která vznikla skládáním transformací prováděných se snímky až do času $t$. Označme ji $I_t$. Tato matice popisuje transformaci, která bude prováděna se snímkem ve videosekvenci v čase $t$. Matici $I_t$ získáme jako

$$I_t = T_t \cdot D\left(I_{t-1}\right),$$

kde $T_t$ je matice transformace v čase $t$ vypočtená podle (14) nebo (15) a $D(I)$ je operátor, který zajišťuje, že $I$ časem zkonverguje k jednotkové matici. Proto jsou jednotlivé koeficienty transformační matice jsou operátorem $D$ upraveny následujícím způsobem: Mějme matici $I$:

$$I = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}.$$

Protože v systému uvažujeme pouze afinní transformace, platí že $a_{20} = a_{21} = 0$, $a_{22} = 1$. Těmito koeficienty se tedy nemusíme dále zabývat. Koeficienty $a_{02}$ a $a_{12}$ popisují posun, respektive jeho $x$-ovou a $y$-ovou složku. Na tyto koeficienty je aplikován lineární filtr podobně jako v [9, 7]:

$$\begin{aligned} a_{02_t} &= c \cdot a_{02_{t-1}} \\ a_{12_t} &= c \cdot a_{12_{t-1}}. \end{aligned}$$

Koeficienty $a_{00}$, $a_{01}$, $a_{10}$ a $a_{11}$ popisují rotační složku pohybu. Při skládání matic popisujících rotace bude výsledná matice opět popisovat rotaci, a to o úhel, který se rovná součtu úhlů původních rotací. Proč tomu tak je, je ukázáno ve vztahu (16):

$$\begin{bmatrix} \cos\alpha & \sin\alpha & a_{02} \\ -\sin\alpha & \cos\alpha & a_{12} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\beta & \sin\beta & b_{02} \\ -\sin\beta & \cos\beta & b_{12} \\ 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} \cos\alpha\cos\beta - \sin\alpha\sin\beta & \cos\alpha\sin\beta + \sin\alpha\cos\beta & \cdots \\ -\sin\alpha\cos\beta - \cos\alpha\sin\beta & -\sin\alpha\sin\beta + \cos\alpha\cos\beta & \cdots \\ 0 & 0 & 1 \end{bmatrix} =$$

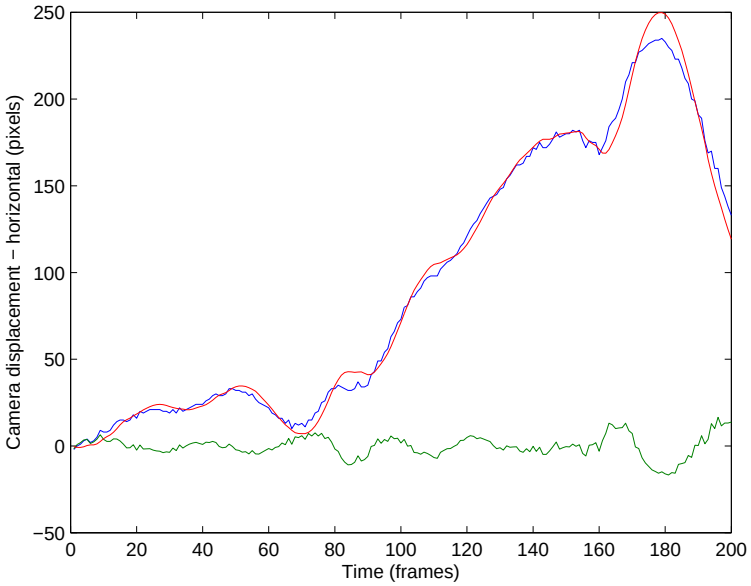
$$\begin{bmatrix} \cos(\alpha+\beta) & \sin(\alpha+\beta) & \cdots \\ -\sin(\alpha+\beta) & \cos(\alpha+\beta) & \cdots \\ 0 & 0 & 1 \end{bmatrix}. \quad (16)$$

Chceme, aby úhel rotace, podobně jako vektor posunutí, zkonvergoval k nule pokud je kamera v klidu. Toho dosáhneme podobně jako v případě posunutí:

$$\alpha_t = c_r \alpha_{t-1}.$$

Výsledná matice, která zachycuje pohyb kamery se tedy vypočítá jako

$$I_t = T_t \cdot \begin{bmatrix} \cos c_R\alpha_{t-1} & \sin c_R\alpha_{t-1} & c_T a_{02_{t-1}} \\ -\sin c_R\alpha_{t-1} & \cos c_R\alpha_{t-1} & c_T a_{12_{t-1}} \\ 0 & 0 & 1 \end{bmatrix},$$




Obrázek 12: Kompenzace pohybu kamery

kde $c_R$ je koeficient, kterým se tlumí rotace kamery a $c_T$ je koeficient, kterým se tlumí posunutí kamery ($0 \leq c_R \leq 1$ a $0 \leq c_T \leq 1$). Když už tedy známe výslednou matici $I_t$, můžeme provést transformaci odpovídajícího snímku z videosekvence podle vztahu (17).

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = I_t \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}. \tag{17}$$

$(x', y')$ jsou zde souřadnice bodu v původním snímku, souřadnice $(x, y)$ jsou souřadnice bodu, na který se tento bod zobrazí.

Ukázku funkce této části systému si můžeme prohlédnout na obrázku 12. Jde opět o ukázku horizontální složky pohybu videosekvence zachycené z jedoucího auta, stejně jako na obrázku 11. Modře je zde znázorněn pohyb kamery vypočtený jako součet horizontálních složek jednotlivých pohybů kamery mezi dvěma snímky. Tento pohyb je tedy nejdříve odfiltrován filtrem typu horní propust. Pohyb po filtraci je obrázku vyznačen zeleně. Tento pohyb je vypočtený jako součet *filtrovaných* horizontálních složek jednotlivých pohybů kamery. Tyto vysokofrekvenční složky pohybu jsou dále z videosekvence odstraněny. Pohyb kamery po odstranění těchto složek znázorňuje červená křivka. Všimněme si, že tato křivka opisuje původní pohyb kamery (na obrázku 12 označen modře), je ovšem daleko plynulejší díky odstranění vysokofrekvenčních složek.

# 4 Provedené optimalizace

V této kapitole budou popsány optimalizace provedené za účelem urychlení. Profilování programu ukázalo, že výpočetně nejnáročnější část systému je vyhledávání bloků. Nejvíce úsilí proto bylo věnováno paralelizaci Bitplane–Matchingu. K tomu bylo využito instrukcí ze sady SSE, díky kterým bylo možno zpracovávat více pixelů obrazu najednou. Program byl dále pomocí OpenMP rozdělen do několika vláken. Oba dva přístupy jsou popsány v následujících podkapitolách.

## 4.1 Paralelizace na SSE

Pro optimalizaci párování bloků pomocí *Bitplane matchingu* a *Sum of Absolute Differences* bylo využito. Intel *Streaming SIMD Extensions* (neboli SSE). Zkratka SIMD pochází z Flynnovy klasifikace architektur počítačů a znamená Single Instruction, Multiple Data streams. Instrukce typu SIMD byly poprvé představeny v procesoru Pentium MMX, tehdy šlo o technologii MMX (Multi Media Extensions). S procesorem Pentium III přišly instrukce SSE, později pak SSE2, SSE3 a SSE4.

Procesory schopné vykonávat instrukce SSE mohou provádět jeden výpočet na více datech najednou. Díky vektorovým instrukcím SSE (architektura SIMD) je možné provádět stejnou instrukci (sčítání, odčítání, bitové XOR) na více číslech najednou během jednoho hodinového cyklu. To přináší významné urychlení pro aplikace zpracovávající velké množství dat. Některé aplikace, kde se SSE používá, zahrnují například:

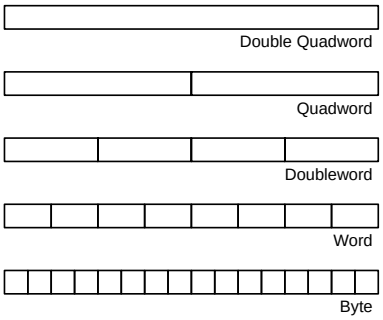
**Zpracování obrazu** — Často je třeba provést nějakou relativně jednoduchou operaci se všemi pixely v obraze. Zpracovávání několika pixelů najednou pomocí vektorových instrukcí zde přináší významné urychlení.

**Zpracování číslicových signálů** — Zde je možné paralelizovat na úrovni zpracovávání jednotlivých vzorků signálu (podobně jako v předchozím případě).

**Hashovací a kódovací algoritmy** — Algoritmy používané pro šifrování a kontrolu integrity dat jako např. CRC (Cyclic Redundancy Check), MD5 (Message Digest Algorithm 5) a SHA (Secure Hash Algorithm) používají jednoduché logické a matematické operace na velkém množství dat.

Instrukce SSE pracují s osmi 128 - bitovými registry označovanými jako XMM0 – XMM7. Data v těchto registrech mohou být interpretována jako hodnoty s pohyblivou desetinnou čárkou nebo jako celočíselné hodnoty. Vzhledem k tomu, že registry mají délku 128 bitů, mohou obsahovat následující data (viz také obrázek 13):

- čtyři 32-bitová čísla s pohyblivou desetinnou čárkou
- dvě 64-bitová čísla s pohyblivou desetinnou čárkou
- jedno 128-bitové celé číslo, Double Quad Word
- dvě 64-bitová celá čísla, Quad Words




Obrázek 13: Celočíselné datové typy v XMM registrech

- čtyři 32-bitová celá čísla, Double Words
- osm 16-bitových celých čísel, Words
- šestnáct 8-bitových celých čísel, Bytes

V této práci budeme dále pracovat pouze s celými čísly. Pro názornost je uspořádání celočíselných datových typů ukázáno na obrázku 13. Instrukce SSE pracují obecně s XMM registry. To, jak budou data v XMM registrech interpretována, závisí konkrétní instrukci. Například instrukce PADDW xmm0, xmm1 sečte osm čísel typu Word v registru xmm0 a osm čísel typu Word v registru xmm1 a výsledek uloží jako osm čísel typu Word do registru xmm0. Instrukce PADDD xmm0, xmm1 sečte čtyři čísla typu Double Word v registru xmm0 a čtyři čísla typu Double Word v registru xmm1 a výsledek uloží jako čtyři čísla typu Double Word do registru xmm0. Vzhledem k tomu, že instrukcí dostupných v jednotlivých verzích SSE je opravdu hodně, dále se omezíme jenom na popis instrukcí, které byly použity v této práci. V následující podkapitole bude popsána optimalizace Bitplane–Matchingu pro SSE2, která byla provedena v rámci této práce. Dále bude stručně popsána implementace Block–Matchingu pomocí sumy absolutních odchylek, která byla převzata z [23] a která byla do systému také zahrnuta. S ohledem na stručnost budou dále popsány pouze zásadní instrukce použité k optimalizaci společně s ukázkami použití těchto instrukcí. Kompletní implementaci funkcí optimalizovaných pro SSE je možné najít v příloze A, příloha A obsahuje pro usnadnění orientace i neoptimalizované verze jednotlivých funkcí.

### 4.1.1 Bitplane–Matching

Jak bylo ukázáno v kapitole 3.1, algoritmus pracuje ve dvou krocích: nejprve vypočítá jednotlivé bitové roviny obrazu, pak provádí vyhledávání a snaží se minimalizovat chybovou hodnotu vypočtenou podle vztahu (11). Oba kroky algoritmu byly optimalizovány pro paralelní výpočet pomocí instrukcí SSE, konkrétní optimalizace budou popsány dále.

**4.1.1.1 Kódování bitových rovin** Nejprve je třeba provést kódování bitových rovin. Abychom zakódovali osm bitových rovin jednoho pixelu obrazu, je třeba jeho hodnotu

```
pixel = src_ptr[i];
dst_ptr[i] = pixel ^ ( pixel >> 1 );
```

Výpis 1: Kódování bitových rovin

```
mov  eax, in_vector
movdqa xmm0, [eax]

movdqa xmm1, xmm0
pand xmm1, xmm7
psrlq xmm1, 0x1
pxor xmm1, xmm0

mov  eax, out_vector
movdqa [eax], xmm1
```

Výpis 2: Kódování bitových rovin pomocí SSE

posunout bitově o jednu doprava a provést operaci XOR s původní hodnotou. Proč tomu tak je je vysvětleno v kapitole 3.1. Původní kód pro kódování bitových rovin je pro úplnost uveden ve výpisu 1.

V programu je nejdříve každý snímek převeden na černobílý, odstíny šedi jsou reprezentovány jako osmibitová čísla. V C++ jde o datový typ `unsigned char`, na obrázku 13 jde o datový typ *Byte*. Kód optimalizovaný pro SSE je ve výpisu 2.

Instrukce MOVDQA načte z paměti 128 bitů, tedy šestnáct pixelů, najednou. V SSE bohužel není instrukce pro bitovou rotaci čísel typu Byte. Proto se nejdříve vynulují nejnižší bity ve všech šestnácti Bytech instrukcí PAND na řádku 6 – v xmm7 je uložena šestnáctkrát za sebou hodnota 0xFE – a pak se provede rotace registru o jednu doprava pomocí instrukce PSRLQ, tedy jako by v registru byly dvě čísla typu Quad Word. Nakonec je proveden XOR a výsledek uložen do paměti. Kompletní implementace optimalizovaných funkcí je možné najít v příloze A.

**4.1.1.2 Párování bloků** Jako časově nejnáročnější operace se ukázalo samotné párování bloků. Při párování bloků postupně procházíme jednotlivé bloky v obraze v čase $t$ a snažíme se najít blok nejpodobnější bloku v obraze v čase $t - 1$. Za nejpodobnější se považuje blok s nejmenší chybovou hodnotou. Chybová hodnota pro dva pixely se spočítá jako bitové XOR hodnot jednotlivých bitových rovin pixelů, chybová hodnota pro dva bloky je pak součet chybových hodnot odpovídajících si pixelů v těchto blocích. Detailnější vysvětlení je možné najít v kapitole 3.1.

Výpočet chybové hodnoty je v C++ realizován kódem v ukázce 3. Kód optimalizovaný pro SSE2 je pak v ukázce 4. Nejdříve jsou načteny hodnoty odpovídajících si pixelů v obou blocích. Hodnoty jsou opět typu Byte, v jednom XMM registru je tedy uloženo šestnáct pixelů. Poté je provedena operace XOR paralelně na všech pixelech najednou,

```
tempErr += (*x_ptr1) ^ (*x_ptr2);
```

Výpis 3: Výpočet chybové hodnoty

```
mov eax, x_ptr1
movdqu xmm0, [eax]

mov eax, x_ptr2
movdqa xmm1, [eax]

pxor xmm0, xmm1
movdqa xmm1, xmm0

punpckhbw xmm0, xmm6
punpcklbw xmm1, xmm6

paddusw xmm7, xmm0
paddusw xmm7, xmm1
```

Výpis 4: Výpočet chybové hodnoty pomocí SSE2

v registru XMM0 je tedy šestnáct chybových hodnot odpovídajících šestnácti pixelům. Tyto hodnoty je teď třeba sečíst. Není však možné sečíst paralelně všech šestnáct čísel najednou, protože by snadno mohlo dojít k přetečení. Proto je k uložení součtů chybových hodnot třeba použít šestnáctibitová čísla. Proto jsou chybové hodnoty nejdříve převedeny na šestnáctibitová čísla pomocí instrukcí PUNPCKLBW a PUNPCKHBW a pak přičteny k celkové chybové hodnotě pro daný blok. Celková chybová hodnota hodnota pro daný blok je uložena v registru XMM7, přesně řečeno jde o osm šestnáctibitových čísel jejichž součet tvoří chybovou hodnotu.

Poté co jsme spočítali chybové hodnoty pro všechny pixely, je třeba sečíst všechna čísla v registru XMM7 abychom získali celkovou chybovou hodnotu. Odpovídající kód je ve výpisu 5. Postup je následující: nejdříve jsou data převedena na čtyři 32-bitová čísla v registru XMM0 a čtyři 32-bitová čísla v registru XMM1, tato jsou paralelně sečtena, pak jsou převedena na dvě 64-bitová čísla v registru XMM0 a dvě v registru XMM1, ta jsou opět paralelně sečtena, a nakonec jsou sečtena dvě výsledná čísla. Výsledek je uložen jako celková chybová hodnota pro daný blok v obrazu. Blok s nejmenší chybovou hodnotou pak určuje lokální pohybový vektor.

### 4.1.2 Sum of Absolute Differences

Do systému byla zahrnuta i implementace párování bloků na základě součtu absolutních odchylek (Sum of Absolute Differences – SAD) podel vztahu 1. Implementace byla převzata z [23]. Ačkoliv párování bloků na základě součtu absolutních odchylek se na první pohled

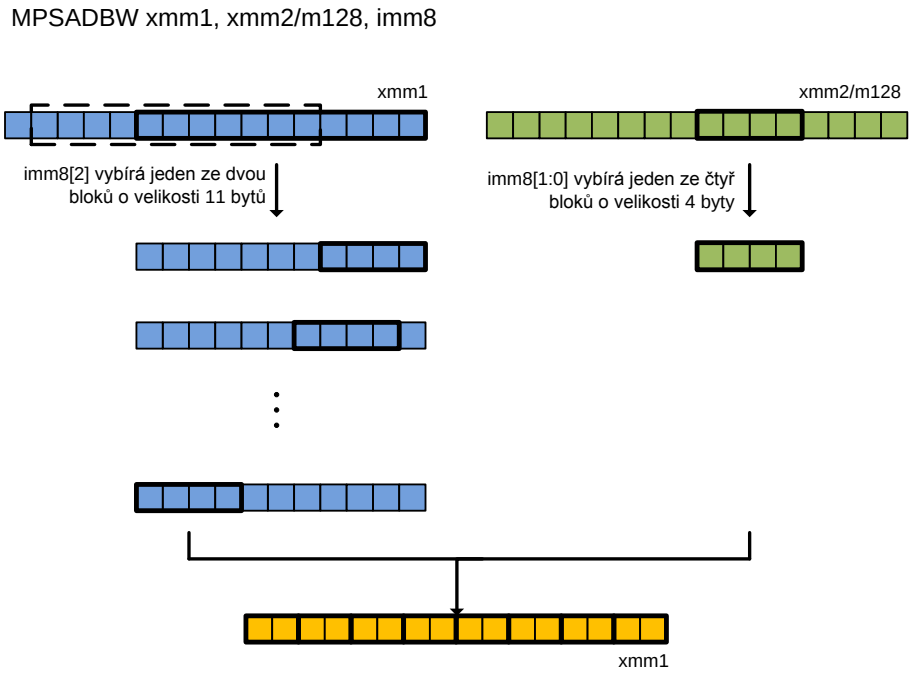
```
movdqa xmm0, xmm7
movdqa xmm1, xmm7

punpcklwd xmm0, xmm6
punpckhwd xmm1, xmm6

paddd xmm0, xmm1
movdqa xmm1, xmm0

punpckldq xmm0, xmm6
punpckhdq xmm1, xmm6

paddq xmm0, xmm1
movdqa xmm1, xmm0

punpcklqdq xmm0, xmm6
punpckhqdq xmm1, xmm6

paddq xmm0, xmm1
pextrd [tempErr], xmm0, 0x0
```

Výpis 5: Výpočet chybové hodnoty pomocí SSE2 – pokračování

jeví jako výpočetně náročnější než v případě Bitplane–Matchingu, díky instrukcím ze sady SSE2 a hlavně SSE4 může být dosaženo výrazného urychlení.

Pro optimalizaci pro SSE2 je výhodné použít instrukci PSADBW. Tato instrukce počítá součet absolutních hodnot rozdílů šestnácti osmibitových čísel v cílovém operandu a šestnácti osmibitových čísel ve zdrojovém operandu. Výsledek je pak uložen v cílovém operandu, a to tak, že součet absolutních hodnot rozdílů spodních osmi osmibitových čísel je uložen jako šestnáctibitové číslo ve spodních dvou bytech cílového operandu a součet absolutních hodnot rozdílů horních osmi osmibitových čísel je uložen v bytech 8 a 9. Cílový operand tedy po provedení instrukce obsahuje dvě šestnáctibitová čísla. Při výpočtu sumy absolutních odchylek pomocí PSADBW postupně procházíme jednotlivé šestnácti-bytové bloky a výsledky sčítáme pomocí PADDUSW (paralelní sčítání osmi šestnáctibitových čísel). Kompletní implementaci funkce optimalizované pro SSE je možné najít v příloze A, příloha A obsahuje pro usnadnění orientace i neoptimalizovanou verzi této funkce.

V sadě SSE4 byla uvedena nová instrukce MPSADBW. Instrukce MPSADBW počítá osm součtů absolutních odchylek najednou, činnost této instrukce je znázorněna na obrázku 14. Formát této instrukce je `MPSADBW xmm0, xmm1/m128, imm8`, instrukce má jako třetí operand osmibitovou konstantu. Pomocí bitů 0 a 1 je vybrán jeden z bloků o velikosti čtyři byty ze zdrojového operandu – na obrázku 14 je tento registr označen zeleně, vybraný blok je zvýrazněn. Bit 2 pak určuje jednu ze dvou částí o velikosti jedenáct bytů ve zdrojovém registru – na obrázku 14 je tento registr označen modře, vybraná část

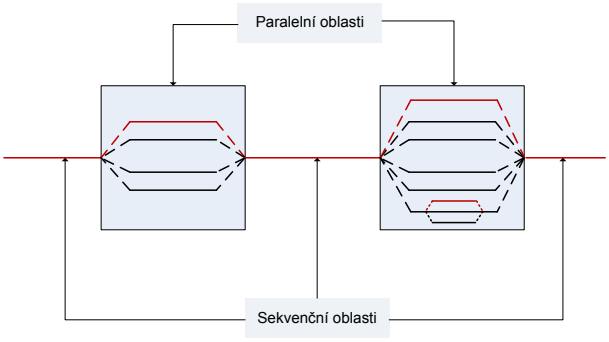



Obrázek 14: Využití instrukce mpsadbw při párování bloků

je opět zvýrazněna. Vybraná část obsahuje osm bloků o velikosti čtyři byty. Instrukce MPSADBW spočítá sumy absolutních odchylek mezi vybraným blokem ze zdrojového registru a osmi bloky o velikosti čtyři byty z vybrané části cílového registru. Výsledkem bude osm hodnot součtů absolutních odchylek, které instrukce uloží do cílového registru jako 16-ti bitové celočíselné hodnoty – na obrázku 14 je cílový registr po provedení instrukce označen žlutě. Při párování bloků pomocí MPSADBW výsledné součty pomocí PADDUSW sčítáme dokud nemáme v cílovém registru chybové hodnoty pro všech osm posunutí. Pomocí PHMINPOSUW pak najdeme nejmenší hodnotu. Jak je ukázáno v [23], pomocí MPSADBW je možné počítat i s bloky jiných velikostí, implementaci funkce pro párování bloků optimalizovanou pro SSE4 je možné najít v příloze A.

## 4.2 Paralelizace pomocí OpenMP

OpenMP usnadňuje vytváření vícevláknových programů v programovacích jazycích Fortran, C a C++. Je to soustava direktiv pro překladač a knihovních funkcí pro paralelní programování, jedná se o standard pro programování počítačů se sdílenou pamětí. Mezi hlavní výhody OpenMP oproti klasickým knihovnám pro tvorbu vláken patří jeho jednoduchost. Hlavní vlákno vytváří podle potřeby za běhu programu několik podvláken, které pak pracují současně (viz ilustrace na obrázku 15, hlavní vlákno je vyznačeno červeně). Nejnáročnější operace v programu většinou probíhají uvnitř smyček, pomocí OpenMP je




Obrázek 15: Paralelní programování s OpenMP

```
#pragma omp for schedule(static)
for (int i = 0; i < numSearchBlocks; i++)
{
   // do block matching for the i−th block...
}
```

Výpis 6: Paralelizace smyčky v OpenMP

možno právě provádění smyčky rozdělit mezi více vláken, a to pouze pomocí jedné direktivy překladače.

Jednoduchý příklad použití OpenMP je uveden v ukázce 6. Smyčka jde přes všechny bloky v obraze, v těle smyčky se pak provádí párování jednoho bloku mezi obrazem v čase $t$ a obrazem v čase $t-1$. Direktiva `#pragma omp for` rozdělí provádění smyčky mezi více vláken, každé vlákno provádí párování několika bloků. Parametr `schedule(static)` říká, že smyčka bude rozdělena staticky, na každé vlákno z celkového počtu $N$ vláken tedy připadne $1/N$ z celkového počtu opakování smyčky. Podobným způsobem byly pomocí OpenMP paralelizovány následující části programu:

**Párování bloků** — jednotlivá vlákna si rozdělí bloky, které se pak budou vyhledávat v následujícím obraze. Každé vlákno tedy provádí vyhledávání několika bloků, přesný počet závisí na počtu vláken a na celkovém počtu bloků zadaném při konfiguraci stabilizátoru.

**Mean–Shift** — vlákna si mezi sebou rozdělí pohybové vektory, pro které budou počítat lokální maximum hustoty, do kterého vektor Mean–Shift dospěje pro ten který pohybový vektor.

**Transformace snímků** — závěrečná transformace snímku z videosekvence za účelem získání stabilizovaného snímku (podle vztahu (17)) se ukázala jako překvapivě časově náročná, proto byla také paralelizována, tentokrát na úrovni jednotlivých bodů v obraze.

# 5 Vyhodnocení výsledků

V této kapitole bude provedeno vyhodnocení dosažených výsledků. Nejprve bude vyhodnocena dosažená úspěšnost stabilizace obrazu. V druhé podkapitole pak bude prezentována dosažená rychlosti výpočtu (připomeňme, že cílem bylo, aby stabilizátor pracoval v reálném čase). Také bude ukázáno urychlení, které přinesly jednotlivé optimalizace, a bude porovnána rychlost jednotlivých pro párování bloků.

## 5.1 Úspěšnost stabilizace

Jako kritérium úspěšnosti stabilizace byla zvolena MSE – *Mean Square Error* vždy mezi dvěma po sobě jdoucími snímky z videosekvence:

$$MSE_t = \frac{1}{MN} \sum_m \sum_n \left(f_t(m,n) - f_{t-1}(m,n)\right)^2 .$$

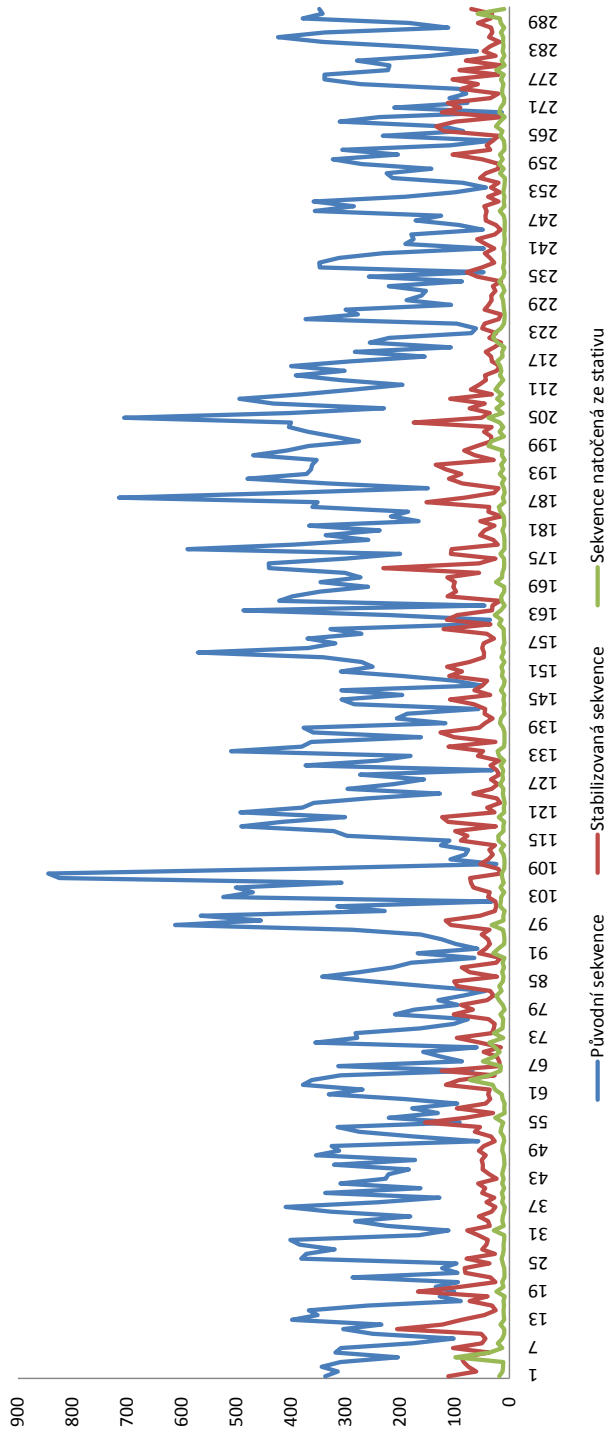
Čím je MSE mezi dvěma snímky menší, tím více jsou si dané snímky podobné. U sekvence obsahující chvění kamery by tedy MSE mělo být vyšší než u stabilizované sekvence.

Pro testování bylo použito sedm testovacích sekvencí: building.avi, tram.avi, flags.avi, lawn.avi, car1.avi, car2.avi a car2.avi. První čtyři sekvence byly natočeny s kamerou namířenou stále na přibližně stejné místo. Nejdříve byla vždy natočena sekvence obsahující chvění kamery, pak byla ta samá scéna natáčena ze stativu. Byly tedy porovnávány tři videosekvence: původní, nestabilizovaná sekvence obsahující chvění kamery, stabilizovaná sekvence a sekvence natočená ze stativu. Dále byl systém testován na třech sekvencích zachycených kamerou umístěnou na čelním skle jedoucího auta: car1.avi, car2.avi a car3.avi. V tomto případě nebyla k dispozici ideální (stabilní) videosekvence, byla tedy porovnávána jen původní sekvence se stabilizovanou. Ukázkové snímky z jednotlivých sekvencí jsou na obrázku 17.

MSE pro jednotlivé dvojice po sobě jdoucích snímků v sekvenci building.avi je zobrazena na obrázku 16. Modře je znázorněna MSE v původní sekvenci, červeně ve stabilizované sekvenci a zeleně pak MSE v sekvenci natočené ze stativu. Jak je vidět, stabilizovaná sekvence má zřetelně nižší MSE než původní, po sobě jdoucí snímky jsou si tedy více podobné. Na obrázku 18 a v tabulce 1 jsou pak vidět průměrné hodnoty MSE pro jednotlivé videosekvence. Je zřejmé, že u statických videosekvencí došlo k výraznému snížení MSE. U videosekvencí natočených z auta už snížení nebylo tak markantní, ačkoliv subjektivně se vizuální kvalita videosekvencí výrazně zlepšila. To lze přičíst tomu, že tyto sekvence obsahují velké množství pohybu – i v ideální, stabilní sekvenci by MSE bylo pravděpodobně dost vysoké.

## 5.2 Urychlení výpočtu

V této kapitole bude popsána dosažená rychlost výpočtu a také urychlení, které přinesly jednotlivé optimalizace. Všechny testy byly prováděny na PC s procesorem Intel Core i7 920 taktovaném na 2.67 GHz s nainstalovaným OS Microsoft Windows XP. Nejdříve




Obrázek 16: MSE nezi jednotlivými dvěma po sobě jdoucími snímky z videosekvence

(a) building.avi

(b) flags.avi

(c) lawn.avi

(d) tram.avi

(e) car1.avi

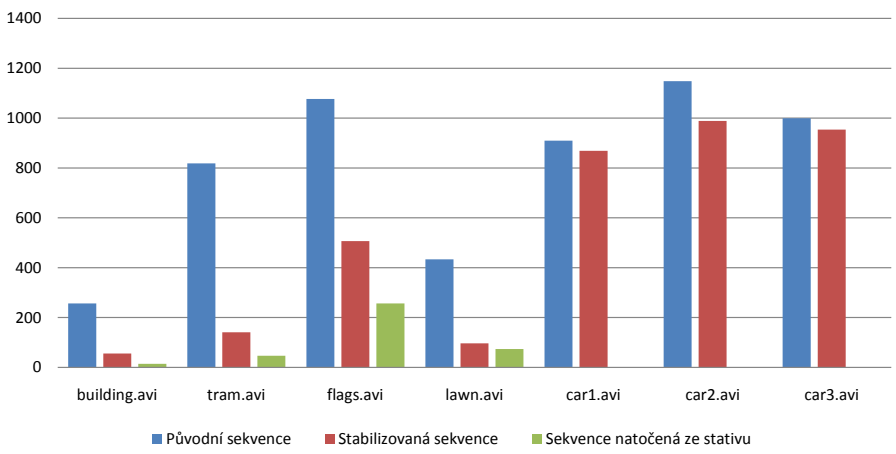

(f) car2.avi


(g) car3.avi

Obrázek 17: Ukázkové snímky z jednotlivých testovacích sekvencí

| sekvence | původní | stabilizovaná | natočená ze stativu |
|---|---|---|---|
| building.avi | 256,7 | 55,4 | 14,5 |
| tram.avi | 818,1 | 140,9 | 46,9 |
| flags.avi | 1076,6 | 506,4 | 256,2 |
| lawn.avi | 433,4 | 96,5 | 73,39 |
| car1.avi | 909,3 | 869,0 | |
| car2.avi | 1147,9 | 988,8 | |
| car3.avi | 999,2 | 953,9 | |

Tabulka 1: Průměrné MSE mezi dvěma po sobě jdoucími snímky v jednotlivých videosekvencích




Obrázek 18: Průměrné MSE v jednotlivých videosekvencích

se podívejme, jaké urychlení přinesla optimalizace jednotlivých algoritmů pro párování bloků na SSE2, resp. SSE4. Byla porovnávána rychlost, s jakou funkce spárují 192 bloků o velikosti $16 \times 16$ pixelů ve dvou po sobě jdoucích obrazech, při měření rychlosti GCBP bylo do měření zahrnuto i kódování bitových rovin. Prohledávána byla vždy oblast $\pm 32$ pixelů v okolí daného bloku. Měřen byl pouze čas (respektive hodinové cykly), které zabralo párování bloků (a kódování bitových rovin v případě GCBP), zbytek programu nebyl do tohoto měření zahrnut.

Dosažené urychlení je ukázáno na obrázku 19, podrobnější výsledky jsou pak v tabulce 2. Nejdříve si všimněme, že neoptimalizovaná verze Bitplane–Matchingu byla mírně rychlejší než neoptimalizovaná verze SAD – asi o 13%. To odpovídá tomu, že GCBP používá pro výpočet chybové hodnoty méně náročnou operaci (GCBP používá XOR zatímco SAD používá absolutní odchylku). Dále je vidět, že optimalizací Bitplane–Matchingu pro SSE2 bylo dosaženo osminásobného urychlení, zatímco urychlení SAD na SSE2 je více než patnáctinásobné.

Podívejme se tedy, proč je rozdíl tak výrazný: pokud zběžně prozkoumáme popis obou implementací v kapitole 4, popřípadě přímo implementace optimalizovaných funkcí v příloze A, zjistíme, že pro výpočet SAD je možné použít instrukce, které počítají více kroků najednou:

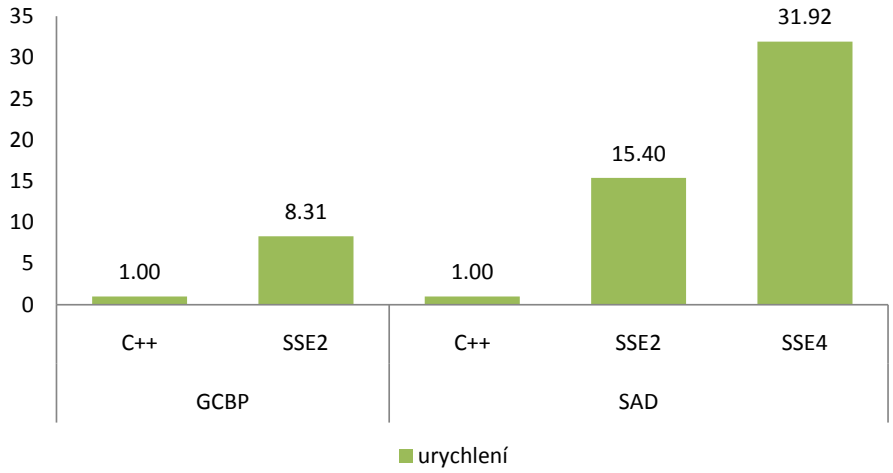
- Pro výpočet chybových hodnot pro šestnáct pixelů pro GCBP je nutné načíst hodnoty z paměti (dvě instrukce), provést bitové XOR (jedna instrukce), pak zkopírovat výsledek do dalšího registru (jedna instrukce), pak obě hodnoty rozbalit (dvě instrukce) a přičíst k celkové chybové hodnotě (dvě instrukce), celkem tedy osm instrukcí na výpočet chybové hodnoty pro šestnáct pixelů.

- Pro výpočet hodnoty SAD pro šestnáct pixelů je třeba nejdříve načíst hodnoty z paměti (dvě instrukce), pak však už stačí jednou zavolat instrukci PSADBW, která spočítá absolutní odchylky, rozbalí je a sečte, a výsledné hodnoty přičíst k celkové chybové hodnotě. Celkem nám tedy stačí čtyři instrukce.

Uvedený pohled je samozřejmě pouze orientační, protože nebere v úvahu latenci ani délku provádění jednotlivých instrukcí. Měl by však ozřejmit to, proč je výpočet SAD na SSE2 oproti GCBP téměř dvakrát rychlejší. U implementace SAD na SSE4 je pak rozdíl ještě markantnější, především díky instrukci MPSADBW. Výpočet pomocí této instrukce probíhá jinak, než výše uvedené výpočty:

- Načteme 24 bytů z prohledávaného obrazu a 16 bytů z bloku, který hledáme (tři instrukce), 16-ti bytový blok pak rozdělíme na čtyři čtyřbytové podbloky, pro každý podblok spočítáme chybové hodnoty pro osm možných posunutí (čtyři instrukce) a výsledky přičteme k výsledným chybovým hodnotám (čtyři instrukce). Tímto jsme spočítali osm chybových hodnot pro osm možných posunutí bloku o velikosti šestnáct bytů.

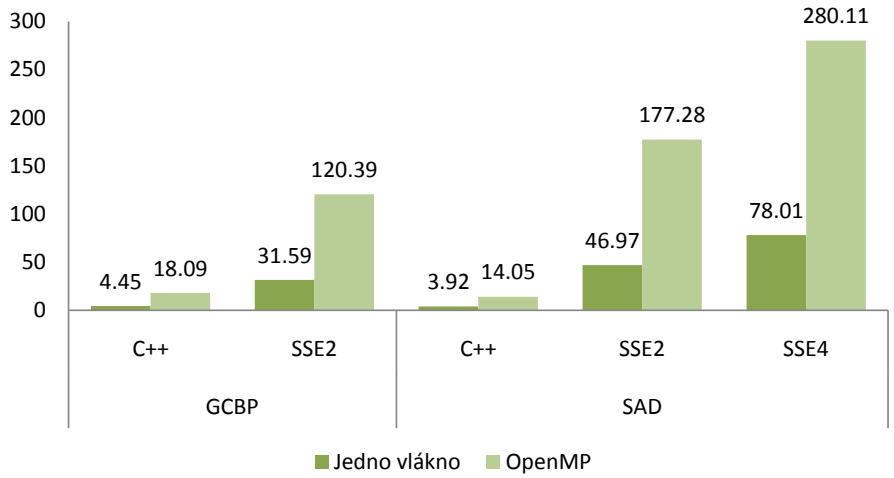
Potřebovali jsme tedy jedenáct instrukcí na to, abychom spočítali sumy 128 absolutních odchylek. Funkce implementovaná na SSE2 by podle odhadu zmíněného výše potřebovala




Obrázek 19: Urychlení porovnávaných algoritmů pomocí SSE

| | | Snímky za sekundu | Urychlení |
|---|---|---|---|
| GCBP | C++ | 4,54 | 1 |
| | SSE2 | 37,74 | 8,31 |
| SAD | C++ | 4 | 1 |
| | SSE2 | 61,7 | 15,39 |
| | SSE4 | 127,92 | 31,92 |

Tabulka 2: Srovnání rychlosti algoritmů pro párování bloků



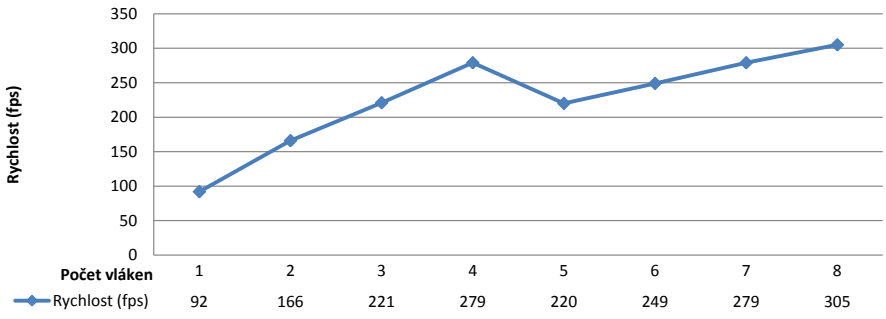

Obrázek 20: Urychlení porovnávaných algoritmů pomocí OpenMP

k tomuto výpočtu 32 instrukcí. Opět zde jde pouze o hrubé nastínění efektivity výpočtu, které slouží jen pro představu o tom, proč bylo naměřené urychlení tak výrazné.

Dále jsme měřili urychlení programu, které přinesla jeho paralelizace pomocí OpenMP. Tentokrát už byl měřen celý program. Procesor Core i7, který byl použit pro testování, má čtyři jádra, díky technologii Hyperthreading však operační systém vidí každé jádro jako dva procesory. Program byl tedy rozdělen do osmi vláken. Hodnoty urychlení jsou vidět na obrázku 20, rychlost programu je zde měřena jako počet stabilizovaných snímků za sekundu. Měřeny byly všechny implementované varianty párování bloků, u všech bylo dosaženo přibližně čtyřnásobného urychlení. Na obrázku 20 je také vidět celkové dosažené urychlení programu: ve verzi bez optimalizace párování bloků a v jednom vláknu pracoval rychlostí $4,45$ snímků za sekundu. Verze optimalizovaná pro SSE4 a paralelizovaná mezi osm vláken pak dosahuje rychlosti mírně přes 280 snímků za sekundu. Celkově tedy bylo dosaženo 63-ti násobného urychlení programu. Měřeno bylo také dosažené urychlení v závislosti na počtu vláken. Měřeny byly varianty od jednoho do osmi vláken, větší počet vláken by vzhledem k vlastnostem procesoru (čtyři jádra, Hyperthreading) neměl smysl. Výsledky tohoto měření jsou na obrázku 21. Jak je vidět, nejrychlejší byl program při rozdělení do osmi vláken. Na obrázku 21 je také vidět, že při použití čtyř vláken je program rychlejší než při použití pěti či šesti vláken.

| | | Jedno vlákno (fps) | OpenMP (fps) | Urychlení |
|---|---|---|---|---|
| GCBP | C++ | 4,45 | 18,09 | 4,06 |
| | SSE2 | 31,59 | 120,39 | 3,81 |
| SAD | C++ | 3,92 | 14,05 | 3,58 |
| | SSE2 | 46,97 | 177,28 | 3,77 |
| | SSE4 | 78,01 | 280,11 | 3,59 |

Tabulka 3: Urychlení porovnávaných algoritmů pomocí OpenMP




Obrázek 21: Rychlost v závislosti na počtu vláken

# 6 Závěr

V této práci byl prezentován systém pro digitální stabilizaci obrazu. Byly zde popsány existující algoritmy pro stabilizaci, z těch byly některé vybrány k vytvoření samotného systému. Dále byl do systému zahrnut algoritmus Mean–Shift, který zde slouží k odstranění chybných pohybových vektorů, a filtrace pohybu kamery IIR filtrem typu horní propust, díky kterému systém kompenzuje pouze chvění kamery, nikoliv úmyslné pohyby kamery.

Úspěšnost stabilizace obrazu dosažená tímto systémem byla měřena na několika videosekvencích. Šlo jednak o sekvence statické, kdy byla kamera namířena na jedno místo, a také sekvence obsahující pohyb kamery – k tomuto měření byly použity sekvence zachycené kamerou umístěnou na čelním skle auta. Jako měřítko úspěšnosti stabilizace bylo zvoleno MSE měřené mezi dvojicemi po sobě jdoucích snímků v daných videosekvencích před a po stabilizaci. U statických videosekvencí došlo k výraznému snížení MSE, jednotlivé snímky ve stabilizovaných sekvencích si tedy byly více podobné než ty v původních sekvencích. U sekvencí zachycených z jedoucího auta už snížení MSE nebylo tak výrazné. To lze však částečně přičíst tomu, že že tyto sekvence obsahovaly velký pohyb kamery. Subjektivní vizuální kvalita stabilizovaných videí byla v obou případech znatelně vyšší.

Implementovaný systém byl profilován, aby se identifikovaly jeho výpočetně nejnáročnější části. Tyto pak byly paralelizovány pomocí SIMD instrukcí a pomocí rozdělení výpočtu mezi několik vláken. Rychlost systému byla měřena na PC s procesorem Intel Core i7, oproti původní neoptimalizované verzi bylo dosaženo 63-ti násobného urychlení programu. Podařilo se docílit toho, že systém je schopen stabilizovat sekvence v reálném čase. Nároky na výpočetní výkon jsou navíc dostatečně nízké na to, aby bylo možné souběžně se stabilizací provádět další analýzu videosekvencí. Systém je tedy možné použít jako předstupeň pro jiné systémy pro analýzu videosekvencí v reálném čase, jako jsou například systémy pro počítání pohybujících se objektů.

# 7 Reference


[1] S. Erturk, *Real-Time Digital Image Stabilization Using Kalman Filters*, Real-Time Imaging 8, 317–328 (2002),doi:10.1006/rtim.2001.0278, dostupné online na http://www.idealibrary.com

[2] A. Amanatiadis, I. Andreadis, A. Gasteratos, N. Kyriakoulis, *A Rotational and Translational Image Stabilization System for Remotely Operated Robots*, IEEE International Workshop on Imaging, Systems and Techniques – IST 2007, Krakow, Poland, May 4–5, 2007

[3] E. Estalayo, L. Salgado, F. Jaureguizar, N. García, *Efficient Image Stabilization and Automatic Target Detection in Aerial FLIR Sequences*, Automatic Target Recognition XVI (Proceedings Volume), Vol. 6234

[4] C. Morimoto, R. Chellapa, *Automatic Digital Image Stabilization* IEEE International Conference on Pattern Recognition, 1996

[5] J.R. Martinez-de Dios, A. Ollero, *A Real-Time Image Stabilization System Based on Fourier-Mellin Transform* ICIAR 2004, LNCS 3211, pp. 376–383, 2004.

[6] Jung-Youp Suk, Gun-Woo Lee, Kuhn-Il Lee, *New Electronic Digital Image Stabilization Algorithm in Wavelet Transform Domain* CIS 2005, Part II, LNAI 3802, pp. 911–916, 2005.

[7] A. C. Brooks, *Real–Time Digital Image Stabilization* EE 420 Image Processing Computer Project Final Paper, MARCH 2003, Electrical Engineering Department, Northwestern University, Evanston, IL 60208 USA

[8] Nam-Joon Kim, Hyuk-Jae Lee, Jae-Beom Lee, *Probabilistic GlobalMotion Estimation Based on Laplacian Two-Bit Plane Matching for Fast Digital Image Stabilization*, Hindawi Publishing Corporation EURASIP Journal on Advances in Signal Processing Volume 2008, Article ID 180582

[9] S. Ko, S. Lee, S. Jeon, and E. Kang, *Fast digital image stabilizer based on gray-coded bit-plane matching*, IEEE Transactions on Consumer Electronics, vol. 45, no. 3, pp. 598-603, Aug. 1999.

[10] Young-Ki Ko, Hyun-Gyu Kim, Jong-Wook Lee, Young-Ro Kim, Hyeong-Cheol Oh and Sung-Jea KO, *New Motion Eestimation Algorithm Based on Bit–Plane Matching and Its VLSI Implementation*, 0-7803-5739-6/99/$10.00(01 999 IEEE.

[11] F. Vella, A. Castorina, M. Mancuso, G. Messina, *Robust Digital Image Stabilization Algorithm Using Block Motion Vectors*, International Conference onConsumer Electronics, 2002. ICCE. 2002 Digest of Technical Papers., pp. 234 - 235, ISBN: 0-7803-7300-6, 2002

[12] Ohyun Kwon, Jeongho Shin, and Joonki Paik, *Video Stabilization Using Kalman Filter and Phase Correlation Matching*, ICIAR 2005, LNCS 3656, pp. 141 – 148, 2005.

[13] Nikolaos Kyriakoulis, Antonios Gasteratos, *A Recursive Fuzzy System for Efficient Digital Image Stabilization*, Hindawi Publishing Corporation, Advances in Fuzzy Systems Volume 2008, Article ID 920615

[14] Shih-Hsuan Yang, Fu-Min Jheng, *An Adaptive Image Stabilization Technique*, IEEE International Conference on Systems, Man, and Cybernetics (SMC2006), Taipei, Taiwan, Oct. 8 - Oct. 11, 2006

[15] S. B. Balakirsky, R. Chelappa, *Perfomance Characterization of Image Stabilization Algorithms* Real-Time Imaging 2 (1996), pp. 297–313

[16] J. P. Dérutin, L. Damez, A. Landrault, *Embedding of a Real Time Image Stabilization Algorithm on SoPC Platform, a Chip Multi-processor Approach*, ACIVS 2008, LNCS 5259, pp. 157–169, 2008

[17] H. Farid, J. B. Woodward, *Video Stabilization and Enhancement*, TR2007-605, Dartmouth College, Computer Science

[18] Pyung Soo Kim, *FIR Filtering Based Image Stabilization Mechanism for Mobile Video Appliances*, J. Zhang, J.-H. He, and Y. Fu (Eds.): CIS 2004, LNCS 3314, pp. 1106–1113, 2004

[19] Chung-Hua Chu, De-Nian Yang, Ming-Syan Chen, *Image Stabilization for 2D Barcode in Handheld Devices* MM'07, September 23–28, 2007, Augsburg, Bavaria, Germany. Copyright 2007 ACM 978-1-59593-701-8/07/0009

[20] T. Fabián, J. Gaura, *Rozšíření výuky ředmětů Analýza obrazu a Digitální zpracování obrazu o úlovhy využívané v rámci střežících systémů*, FRVŠ 1547/2009/G1

[21] P. Doubek, *Mean-Shift segmentace*, dostupné online: `http://cmp.felk.cvut.cz/cmp/courses/ZS1/Cviceni/cv4/meanshift.pdf`

[22] O. Tuzel, P. Meer, *Mean Shift Clustering*, dostupné online: `http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/TUZEL1/MeanShift.pdf`

[23] Kiefer Kuah, *Motion Estimation with Intel® Streaming SIMD Extensions 4 (Intel® SSE4)*, Intel Software Solutions Group, dostupné online: `http://software.intel.com/en-us/articles/ motion-estimation-with-intel-streaming-simd-extensions-4-intel-sse4/`

# A Optimalizované funkce

```
void GCBP::getGrayCodedBitPlane(IplImage *src, IplImage *
   dst)
{
   uchar *src_ptr;
   uchar *dst_ptr;
  uchar pixel;
  int i, j;

   cvSetZero(dst);
   for (j = 0; j < src->height; j++)
   {
      src_ptr = (uchar*) (src->imageData + j * src->
         widthStep);
      dst_ptr = (uchar*) (dst->imageData + j * dst->
         widthStep);

      for (i = 0; i < src->width; i++)
      {
       pixel = src_ptr[i];
       dst_ptr[i] = pixel ^ ( pixel >> 1 );
      }
   }
}
```

Výpis 7: Kódování bitových rovin

```
void GCBP_SSE::getGrayCodedBitPlane(IplImage *src, IplImage
    *dst)
{
   uchar *src_ptr;
   uchar *in_vector;
  uchar *dst_ptr, *out_vector;
  int i, j;

   cvSetZero(dst);

  uchar *mask = SSE_RotMask;
  __asm {
    mov eax, [mask]
    movdqa xmm7, [eax]
  }
```

```
   for (j = 0; j < src->height; j++)
   {
      src_ptr = (uchar*) (src->imageData + j * src->
         widthStep);
      dst_ptr = (uchar*) (dst->imageData + j * dst->
         widthStep);

      for (i = 0; i < src->width; i += 16)
      {
         in_vector = src_ptr + i;
         out_vector = dst_ptr + i;

       __asm {
         mov  eax, in_vector
         movdqa xmm0, [eax]

              movdqa xmm1, xmm0
            pand xmm1, xmm7
         psrlq xmm1, 0x1
            pxor xmm1, xmm0

         mov  eax, out_vector
            movdqa [eax], xmm1
       }
      }
   }
}
```

Výpis 8: Kódování bitových rovin na SSE2

```
int BlockMatcherGCBP_SSE2::blockMatch(const unsigned char*
   refFrame, int stepBytesRF,
                const unsigned char* curBlock, int
                   stepBytesCB,
                int* matchBlock,
                int frameWidth, int frameHeight)
{
  unsigned int lowSum = UINT_MAX;
  unsigned int temSum = 0;
  int blockHeight = 16;
  int blockWidth = 16;
  const unsigned char *pRef, *pCur;
  __m128i s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11,
      s12, s13, s14, s15, s16, sZERO, sXOR;
```

```
pCur = curBlock;
s0 = _mm_loadu_si128((__m128i*)pCur);
s1 = _mm_loadu_si128((__m128i*)(pCur+stepBytesCB));
s2 = _mm_loadu_si128((__m128i*)(pCur+2*stepBytesCB));
s3 = _mm_loadu_si128((__m128i*)(pCur+3*stepBytesCB));
s4 = _mm_loadu_si128((__m128i*)(pCur+4*stepBytesCB));
s5 = _mm_loadu_si128((__m128i*)(pCur+5*stepBytesCB));
s6 = _mm_loadu_si128((__m128i*)(pCur+6*stepBytesCB));
s7 = _mm_loadu_si128((__m128i*)(pCur+7*stepBytesCB));
s8 = _mm_loadu_si128((__m128i*)(pCur+8*stepBytesCB));
s9 = _mm_loadu_si128((__m128i*)(pCur+9*stepBytesCB));
s10 = _mm_loadu_si128((__m128i*)(pCur+10*stepBytesCB));
s11 = _mm_loadu_si128((__m128i*)(pCur+11*stepBytesCB));
s12 = _mm_loadu_si128((__m128i*)(pCur+12*stepBytesCB));
s13 = _mm_loadu_si128((__m128i*)(pCur+13*stepBytesCB));
s14 = _mm_loadu_si128((__m128i*)(pCur+14*stepBytesCB));
s15 = _mm_loadu_si128((__m128i*)(pCur+15*stepBytesCB));

sZERO = _mm_setzero_si128();

for (int i=0; i<=frameHeight-blockHeight; i++)
{
  for (int j=0; j<=frameWidth-blockWidth; j++)
  {
    pRef = refFrame+i*stepBytesRF+j;
    s16 = _mm_setzero_si128();

    sXOR = _mm_xor_si128(s0, _mm_loadu_si128((__m128i*)
       pRef));
    s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
       _mm_unpacklo_epi8(sXOR, sZERO),
         _mm_unpackhi_epi8(sXOR, sZERO) ) );

    sXOR = _mm_xor_si128(s1, _mm_loadu_si128((__m128i*)(
       pRef+stepBytesRF)));
    s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
       _mm_unpacklo_epi8(sXOR, sZERO),
         _mm_unpackhi_epi8(sXOR, sZERO) ) );

    sXOR = _mm_xor_si128(s2, _mm_loadu_si128((__m128i*)(
       pRef+2*stepBytesRF)));
```

```
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s3, _mm_loadu_si128((__m128i*)(
   pRef+3*stepBytesRF)));
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s4, _mm_loadu_si128((__m128i*)(
   pRef+4*stepBytesRF)));
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s5, _mm_loadu_si128((__m128i*)(
   pRef+5*stepBytesRF)));
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s6, _mm_loadu_si128((__m128i*)(
   pRef+6*stepBytesRF)));
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s7, _mm_loadu_si128((__m128i*)(
   pRef+7*stepBytesRF)));
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s8, _mm_loadu_si128((__m128i*)(
   pRef+8*stepBytesRF)));
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s9, _mm_loadu_si128((__m128i*)(
   pRef+9*stepBytesRF)));
```

```
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s10, _mm_loadu_si128((__m128i*)(
   pRef+10*stepBytesRF)));
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s11, _mm_loadu_si128((__m128i*)(
   pRef+11*stepBytesRF)));
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s12, _mm_loadu_si128((__m128i*)(
   pRef+12*stepBytesRF)));
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s13, _mm_loadu_si128((__m128i*)(
   pRef+13*stepBytesRF)));
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s14, _mm_loadu_si128((__m128i*)(
   pRef+14*stepBytesRF)));
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

sXOR = _mm_xor_si128(s15, _mm_loadu_si128((__m128i*)(
   pRef+15*stepBytesRF)));
s16 = _mm_adds_epu16(s16,_mm_adds_epu16(
   _mm_unpacklo_epi8(sXOR, sZERO),
     _mm_unpackhi_epi8(sXOR, sZERO) ) );

__m128i sumDW = _mm_add_epi32(_mm_unpacklo_epi16(s16,
    sZERO), _mm_unpackhi_epi16(s16, sZERO));
```

```
        __m128i sumQW = _mm_add_epi64(_mm_unpacklo_epi32(dw,
           sZERO), _mm_unpackhi_epi32(dw, sZERO));
        temSum = _mm_extract_epi16(qw, 0) + _mm_extract_epi16
           (qw, 4);

        if (temSum < lowSum)
        {
          lowSum = temSum;
          *matchBlock = j;
          *(matchBlock+1) = i;
        }
      }
    }

    return 0;
}
```

Výpis 9: Implementace GCBP na SSE2

```
int BlockMatcherMAD::blockMatch(const unsigned char*
   refFrame, int stepBytesRF /*stride*/,
                  const unsigned char* curBlock, int
                     stepBytesCB /*stride*/,
                  int* matchBlock /*results*/,
                  int frameWidth, int frameHeight)
{
  int lowSum = INT_MAX;
  int temSum = 0;
  int blockHeight = 16;
  int blockWidth = 16;
  const unsigned char *pRef, *pCur;

  for (int i=0; i<=frameHeight-blockHeight; i++)
  {
    for (int j=0; j<=frameWidth-blockWidth; j++)
    {
      temSum = 0;
      pCur = curBlock;
      pRef = refFrame+i*stepBytesRF+j;

        for (int k=0; k<blockHeight; k++)
        {
          for (int l=0; l<blockWidth; l++)
          {
```

```
            temSum += labs(*pRef-*pCur);
            pCur++;
            pRef++;
          }
          pCur+=stepBytesCB-blockWidth;
          pRef+=stepBytesRF-blockWidth;
        }

        if (temSum < lowSum)
        {
          lowSum = temSum;
          *matchBlock = j;
          *(matchBlock+1) = i;
        }
      }
    }

  return 0;
}
```

Výpis 10: Původní neoptimalizovaná implementace SAD

```
int BlockMatcherMAD_SSE2::blockMatch(const unsigned char*
   refFrame, int stepBytesRF,
                    const unsigned char* curBlock, int
                       stepBytesCB,
                    int* matchBlock,
                    int frameWidth, int frameHeight)
{
  unsigned int lowSum = UINT_MAX;
  unsigned int temSum = 0;
  int blockHeight = 16;
  int blockWidth = 16;
  const unsigned char *pRef, *pCur;
  __m128i s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11,
      s12, s13, s14, s15, s16;

  pCur = curBlock;
  s0 = _mm_loadu_si128((__m128i*)pCur);
  s1 = _mm_loadu_si128((__m128i*)(pCur+stepBytesCB));
  s2 = _mm_loadu_si128((__m128i*)(pCur+2*stepBytesCB));
  s3 = _mm_loadu_si128((__m128i*)(pCur+3*stepBytesCB));
  s4 = _mm_loadu_si128((__m128i*)(pCur+4*stepBytesCB));
  s5 = _mm_loadu_si128((__m128i*)(pCur+5*stepBytesCB));
```

```
s6 = _mm_loadu_si128((__m128i*)(pCur+6*stepBytesCB));
s7 = _mm_loadu_si128((__m128i*)(pCur+7*stepBytesCB));
s8 = _mm_loadu_si128((__m128i*)(pCur+8*stepBytesCB));
s9 = _mm_loadu_si128((__m128i*)(pCur+9*stepBytesCB));
s10 = _mm_loadu_si128((__m128i*)(pCur+10*stepBytesCB));
s11 = _mm_loadu_si128((__m128i*)(pCur+11*stepBytesCB));
s12 = _mm_loadu_si128((__m128i*)(pCur+12*stepBytesCB));
s13 = _mm_loadu_si128((__m128i*)(pCur+13*stepBytesCB));
s14 = _mm_loadu_si128((__m128i*)(pCur+14*stepBytesCB));
s15 = _mm_loadu_si128((__m128i*)(pCur+15*stepBytesCB));

for (int i=0; i<=frameHeight-blockHeight; i++)
{
  for (int j=0; j<=frameWidth-blockWidth; j++)
  {
    pRef = refFrame+i*stepBytesRF+j;

    s16 = _mm_sad_epu8(s0, _mm_loadu_si128((__m128i*)pRef
       ));
    s16 = _mm_adds_epu16(s16,
           _mm_sad_epu8(s1,
           _mm_loadu_si128((__m128i*)(pRef+stepBytesRF)))
              );
    s16 = _mm_adds_epu16(s16,
           _mm_sad_epu8(s2,
           _mm_loadu_si128((__m128i*)(pRef+2*stepBytesRF)
              )));
    s16 = _mm_adds_epu16(s16,
           _mm_sad_epu8(s3,
           _mm_loadu_si128((__m128i*)(pRef+3*stepBytesRF)
              )));
    s16 = _mm_adds_epu16(s16,
           _mm_sad_epu8(s4,
           _mm_loadu_si128((__m128i*)(pRef+4*stepBytesRF)
              )));
    s16 = _mm_adds_epu16(s16,
           _mm_sad_epu8(s5,
           _mm_loadu_si128((__m128i*)(pRef+5*stepBytesRF)
              )));
    s16 = _mm_adds_epu16(s16,
           _mm_sad_epu8(s6,
           _mm_loadu_si128((__m128i*)(pRef+6*stepBytesRF)
              )));
```

```
s16 = _mm_adds_epu16(s16,
       _mm_sad_epu8(s7,
       _mm_loadu_si128((__m128i*)(pRef+7*stepBytesRF)
          )));
s16 = _mm_adds_epu16(s16,
       _mm_sad_epu8(s8,
       _mm_loadu_si128((__m128i*)(pRef+8*stepBytesRF)
          )));
s16 = _mm_adds_epu16(s16,
       _mm_sad_epu8(s9,
       _mm_loadu_si128((__m128i*)(pRef+9*stepBytesRF)
          )));
s16 = _mm_adds_epu16(s16,
       _mm_sad_epu8(s10,
       _mm_loadu_si128((__m128i*)(pRef+10*stepBytesRF
          ))));
s16 = _mm_adds_epu16(s16,
       _mm_sad_epu8(s11,
       _mm_loadu_si128((__m128i*)(pRef+11*stepBytesRF
          ))));
s16 = _mm_adds_epu16(s16,
       _mm_sad_epu8(s12,
       _mm_loadu_si128((__m128i*)(pRef+12*stepBytesRF
          ))));
s16 = _mm_adds_epu16(s16,
       _mm_sad_epu8(s13,
       _mm_loadu_si128((__m128i*)(pRef+13*stepBytesRF
          ))));
s16 = _mm_adds_epu16(s16,
       _mm_sad_epu8(s14,
       _mm_loadu_si128((__m128i*)(pRef+14*stepBytesRF
          ))));
s16 = _mm_adds_epu16(s16,
       _mm_sad_epu8(s15,
       _mm_loadu_si128((__m128i*)(pRef+15*stepBytesRF
          ))));
temSum = _mm_extract_epi16(s16,0) + _mm_extract_epi16
   (s16,4);

if (temSum < lowSum)
{
  lowSum = temSum;
  *matchBlock = j;
```

```
            *(matchBlock+1) = i;
          }
        }
      }

      return 0;
    }
```

Výpis 11: Implementace SAD pro SSE2

```
int BlockMatcherMAD_SSE4::blockMatch(const unsigned char*
    refFrame, int stepBytesRF,
                        const unsigned char* curBlock, int
                            stepBytesCB,
                        int* matchBlock,
                        int frameWidth, int frameHeight)
{
  unsigned int lowSum = UINT_MAX;
  unsigned int temSum = 0;
  int blockHeight = 16;
  int blockWidth = 16;
  int k;
  const unsigned char *pRef, *pCur;
  __m128i s0, s1, s2, s3, s4, s5, s6, s7;

  for (int i=0; i<=frameHeight-blockHeight; i++)
  {
    int j=0;
    for (j=0; j<=frameWidth-24; j+=8)
    {
      pCur = curBlock;
      pRef = refFrame+i*stepBytesRF+j;
      s3 = _mm_setzero_si128();
      s4 = _mm_setzero_si128();
      s5 = _mm_setzero_si128();
      s6 = _mm_setzero_si128();

      for (k=0; k<blockHeight; k++)
      {
        s0 = _mm_loadu_si128((__m128i*)pRef);
        s1 = _mm_loadu_si128((__m128i*)(pRef+8));
        s2 = _mm_loadu_si128((__m128i*)pCur);
        s3 = _mm_adds_epu16(s3, _mm_mpsadbw_epu8(s0, s2,
            0));
```

```
      s4 = _mm_adds_epu16(s4, _mm_mpsadbw_epu8(s0, s2,
         5));
      s5 = _mm_adds_epu16(s5, _mm_mpsadbw_epu8(s1, s2,
         2));
      s6 = _mm_adds_epu16(s6, _mm_mpsadbw_epu8(s1, s2,
         7));
      pCur+=stepBytesCB;
      pRef+=stepBytesRF;
    }
    s7 = _mm_adds_epu16(_mm_adds_epu16(s3, s4),
       _mm_adds_epu16(s5, s6));
    s7 = _mm_minpos_epu16(s7);
    temSum = _mm_extract_epi16(s7,0);

    if (temSum < lowSum)
    {
      lowSum = temSum;
      k = _mm_extract_epi16(s7,1);
      *matchBlock = j+k;
      *(matchBlock+1) = i;
    }
  }

  for (; j<=frameWidth-blockWidth; j++)
  {
    pCur = curBlock;
    pRef = refFrame+i*stepBytesRF+j;

    s2 = _mm_setzero_si128();

    for (k=0; k<blockHeight; k++)
    {
      s0 = _mm_loadu_si128((__m128i*)pRef);
      s1 = _mm_loadu_si128((__m128i*)pCur);
      s2 = _mm_adds_epu16(s2, _mm_sad_epu8(s0, s1));

      pCur+=stepBytesCB;
      pRef+=stepBytesRF;
    }

    temSum = _mm_extract_epi16(s2,0) + _mm_extract_epi16(
       s2,4);
```

```
            if (temSum < lowSum)
            {
               lowSum = temSum;
               *matchBlock = j;
               *(matchBlock+1) = i;
            }
         }
      }

      return 0;
   }
```

Výpis 12: Implementace SAD pro SSE4